I'm not a bot

Modified on: Tue, Nov 1, 2022 at 11:54 AM This article covers Date/Time Formats.Table of ContentsDate and time formats are specified by date and time pattern strings, within which unquoted letters, from ‘A’ to ‘Z’ and ‘a’ to ‘z,’ are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (’) to avoid interpretation. All other characters are not interpreted; they are simply copied into the output string during formatting, or matched against the input string during parsing. The following pattern letters are defined (all other characters, from ‘A’ to ‘Z’ and ‘a’ to ‘z,’ are reserved):ReferencesLetterDate or Time ComponentPresentationExamplesGEraTextADyYearYear1996, 96MMonth in yearMonthJuly, Jul, 07wWeek in yearNumber27WWeek in monthNumber2DDay in yearNumber189dDay in monthNumber10FDay of week in monthNumber2EDay in weekTextTuesday, TueaAM/PMTextPMHHour in day (0-23)Number0kHour in day (1-24)Number24hHour in am/pm (1-12)Number12KHour in am/pm (0-11)Number0mMinute in hourNumber30sSecond in minuteNumber55SMillisecondNumber978zTime zoneGeneral time zonePacific Standard Time, PST, GMT -08:00ZTime zoneRFC 822 time zone-0800Pattern letters are usually repeated, as their number determines the exact presentation:TextFor formatting, if the number of pattern letters is four or more, the full form is used; otherwise, a short or abbreviated form is used, if available. For parsing, both forms are accepted, regardless of the number of pattern letters.NumberFor formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.YearFor formatting, if the number of pattern letters is two, the year is truncated to two digits; otherwise, it is interpreted as a NumberFormat.For parsing, if the number of pattern letters is more than two, the year is interpreted literally, regardless of the number of digits; therefore, using the pattern “MM/dd/yyyy”, “01/11/12″ parses to Jan 11, 12 A.D. For parsing with the abbreviated year pattern (“y” or “yy”), SimpleDateFormat must interpret the abbreviated year, relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the SimpleDateFormat instance is created. For example, using a pattern of “MM/dd/yy”; and a SimpleDateFormat instance created on Jan 1, 1997, the string “01/11/12″ would be interpreted as Jan 11, 2012, while the string “05/04/64″ would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, as defined by Character.isDigit(char), will be parsed into the default century. Any other numeric string, such as a one-digit string, a three- or more digit string, or a two-digit string that does not consist entirely of digits (e.g., “-1″), is interpreted literally. Therefore, “01/02/3″ or “01/02/003″ are parsed using the same pattern as Jan 2, 3 AD. Likewise, “01/02/-3″; is parsed as Jan 2, 4 BC.MonthIf the number of pattern letters is three or more, the month is interpreted as text; otherwise, it is interpreted as a number.General Time ZoneTime zones are interpreted as text when they have names. For time zones representing a GMT offset value, the following syntax is used:GMTOffsetTimeZone: GMT Sign Hours: MinutesSign: one of +-Hours: Digit Digit DigitMinutes: Digit DigitDigit: one of 0 1 2 3 4 5 6 7 8 9Hours: must be between 0 and 23Minutes: must be between 00 and 59The format is locale independent, and digits must be taken from the Basic Latin block of the Unicode standard.RFC 822 Time ZoneRFC 822 time zones are also acceptable for parsing. For formatting, the RFC 822 4-digit time zone format is used:RFC822TimeZone: Sign TwoDigitHours MinutesTwoDigitHours: Digit DigitTwoDigitHours must be between 00 and 23Other Time ZonesOther definitions are as those for general time zones. For parsing, general time zones are also accepted. SimpleDateFormat also supports localized date and time pattern strings. In these strings, the pattern letters described above may be replaced with other, locale dependent pattern letters. SimpleDateFormat does not deal with the localization of text, other than the pattern letters; localization of text is dependent upon the client of the class.ExamplesThe following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific time zone.Date and Time PatternResultyyyy.MM.dd G HH:mm:ss2001.07.04 AD 12:08:56EEE MMM d, yyWed Jul 4, 01hh a, zzzz12 PM, Pacific Daylight TimeK;mm a, z0:08 PM, PDTyyyyy.MMMMM.dd GGG hh:mm aaa02001.July.04 AD 12:08 PMEEE, d MMM yyyy HH:mm:ss ZWed, 4 Jul 2001 12:08:56 -0700yyMMddHHmmssZ010704120856-0700Please visit the Fountayn Contact Information page. Did you find it helpful? Yes No All Implemented Interfaces: Serializable, Cloneable public class SimpleDateFormat extends DateFormat SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. It allows for formatting (date → text), parsing (text → date), and normalization. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting. However, you are encouraged to create a date-time formatter with either getTimeInstance, getDateInstance, or getDateTimeInstance in DateFormat. Each of these class methods can return a date/time formatter initialized with a default format pattern. You may modify the format pattern using the applyPattern methods as desired. For more information on using these methods, see DateFormat. Date and Time Patterns Date and time formats are specified by date and time pattern strings. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. "'" represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing. The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved): Letter Date or Time Component Presentation Examples G Era designator Text AD y Year Year 1996; 96 Y Week year Year 2009; 09 M Month in year (context sensitive) Month July; Jul; 07 L Month in year (standalone form) Month July; Jul; 07 w Week in year Number 27 W Week in month Number 2 D Day in year Number 189 d Day in month Number 10 F Day of week in month Number 2 E Day name in week Text Tuesday; Tue u Day number of week (1 = Monday, ..., 7 = Sunday) Number 1 a Am/pm marker Text PM H Hour in day (0-23) Number 0 k Hour in day (1-24) Number 24 K Hour in am/pm (0-11) Number 0 h Hour in am/pm (1-12) Number 12 m Minute in hour Number 30 s Second in minute Number 55 S Millisecond Number 978 z Time zone General time zone Pacific Standard Time; PST; GMT-08:00 Z Time zone RFC 822 time zone -0800 X Time zone ISO 8601 time zone -08; -0800; -08:00 Pattern letters are usually repeated, as their number determines the exact presentation: Text: For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters. Number: For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields. Year: If the formatter's Calendar is the Gregorian calendar, the following rules are applied. For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a number. For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern "MM/dd/yyyy", "01/11/12" parses to Jan 11, 12 A.D. For parsing with the abbreviated year pattern ("y" or "yy"), SimpleDateFormat must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the SimpleDateFormat instance is created. For example, using a pattern of "MM/dd/yy" and a SimpleDateFormat instance created on Jan 1, 1997, the string "01/11/12" would be interpreted as Jan 11, 2012 while the string "05/04/64" would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, as defined by Character.isDigit(char), will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that isn't all digits (for example, "-1"), is interpreted literally. So "01/02/3" or "01/02/003" are parsed, using the same pattern, as Jan 2, 3 AD. Likewise, "01/02/-3" is parsed as Jan 2, 4 BC. Otherwise, calendar system specific forms are applied. For both formatting and parsing, if the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar specific short or abbreviated form is used. If week year 'Y' is specified and the calendar doesn't support any week years, the calendar year ('y') is used instead. The support of week years can be tested with a call to getCalendar().isWeekDateSupported(). Month: If the number of pattern letters is 3 or more, the month is interpreted as text; otherwise, it is interpreted as a number. General time zone: Time zones are interpreted as text if they have names. For time zones representing a GMT offset value, the following syntax is used: GMTOffsetTimeZone: GMT Sign Hours : Minutes Sign: one of + - Hours: Digit Digit Digit Minutes: Digit Digit Digit: one of 0 1 2 3 4 5 6 7 8 9 Hours must be between 0 and 23, and Minutes must be between 00 and 59. The format is locale independent and digits must be taken from the Basic Latin block of the Unicode standard. For parsing, RFC 822 time zones are also accepted. RFC 822 time zone: For formatting, the RFC 822 4-digit time zone format is used: RFC822TimeZone: Sign TwoDigitHours Minutes TwoDigitHours: Digit Digit TwoDigitHours must be between 00 and 23. Other definitions are as for general time zones. For parsing, general time zones are also accepted. ISO 8601 Time zone: The number of pattern letters designates the format for both formatting and parsing as follows: ISO8601TimeZone: OneLetterISO8601TimeZone TwoLetterISO8601TimeZone ThreeLetterISO8601TimeZone OneLetterISO8601TimeZone: Sign TwoDigitHours Z TwoLetterISO8601TimeZone: Sign TwoDigitHours Minutes Z ThreeLetterISO8601TimeZone: Sign TwoDigitHours : Minutes Z Other definitions are as for general time zones or RFC 822 time zones. For formatting, if the offset value from GMT is 0, "Z" is produced. If the number of pattern letters is 1, any fraction of an hour is ignored. For example, if the pattern is "X" and the time zone is "GMT+05:30", "+05" is produced. For parsing, "Z" is parsed as the UTC time zone designator. General time zones are not accepted. If the number of pattern letters is 4 or more, IllegalArgumentException is thrown when constructing a SimpleDateFormat or applying a pattern. SimpleDateFormat also supports localized date and time pattern strings. In these strings, the pattern letters described above may be replaced with other, locale dependent, pattern letters. SimpleDateFormat does not deal with the localization of text other than the pattern letters; that's up to the client of the class. Examples The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone. Date and Time Pattern Result "yyyy.MM.dd G 'at' HH:mm:ss z" 2001.07.04 AD at 12:08:56 PDT "EEE, MMM d, "yy" Wed, Jul 4, '01 "h:mm a" 12:08 PM "hh 'o''clock' a, zzzz" 12 o'clock PM, Pacific Daylight Time "K:mm a, z" 0:08 PM, PDT "yyyyy.MMMMM.dd GGG hh:mm aaa" 02001.July.04 AD 12:08 PM "EEE, d MMM yyyy HH:mm:ss Z" Wed, 4 Jul 2001 12:08:56 -0700 "yyMMddHHmmssZ" 010704120856-0700 "yyyy-MM-dd'T'HH:mm:ss.SSSZ" 2001-07-04T12:08:56.235-0700 "yyyy-MM-dd'T'HH:mm:ss.SSSXXX" 2001-07-04T12:08:56.235-07:00 "YYYY-'W'ww-u" 2001-W27-3 Synchronization Date formats are not synchronized. It is recommended to create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally. See Also: Java Tutorial, Calendar, TimeZone, DateFormat, DateFormatSymbols, Serialized Form AM_PM_FIELD, calendar, DATE_FIELD, DAY_OF_WEEK_FIELD, DAY_OF_WEEK_IN_MONTH_FIELD, DAY_OF_YEAR_FIELD, DEFAULT, ERA_FIELD, FULL, HOUR_OF_DAY0_FIELD, HOUR_OF_DAY1_FIELD, HOUR0_FIELD, HOUR1_FIELD, LONG, MEDIUM, MILLISECOND_FIELD, MINUTE_FIELD, MONTH_FIELD, numberFormat, SECOND_FIELD, SHORT, TIMEZONE_FIELD, WEEK_OF_MONTH_FIELD, WEEK_OF_YEAR_FIELD, YEAR_FIELD format, format, getAvailableLocales, getCalendar, getDateInstance, getDateInstance, getDateInstance, getDateTimeInstance, getDateTimeInstance, getDateTimeInstance, getDateTimeInstance, getInstance, getNumberFormat, getTimeInstance, getTimeInstance, getTimeInstance, getTimeZone, isLenient, parse, parseObject, setCalendar, setLenient, setNumberFormat, setTimeZone finalize, getClass, notify, notifyAll, toString, wait, wait, wait public SimpleDateFormat() Constructs a SimpleDateFormat using the default pattern and date format symbols for the default FORMAT locale. Note: This constructor may not support all locales. For full coverage, use the factory methods in the DateFormat class. public SimpleDateFormat(String pattern, Locale locale) Constructs a SimpleDateFormat using the given pattern and the default date format symbols for the given locale. Note: This constructor may not support all locales. For full coverage, use the factory methods in the DateFormat class. Parameters: pattern - the pattern describing the date and time format locale - the locale whose date format symbols should be used Throws: NullPointerException - if the given pattern or locale is null IllegalArgumentException - if the given pattern is invalid public SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols) Constructs a SimpleDateFormat using the given pattern and date format symbols. Parameters: pattern - the pattern describing the date and time format formatSymbols - the date format symbols to be used for formatting Throws: NullPointerException - if the given pattern or formatSymbols is null IllegalArgumentException - if the given pattern is invalid public void set2DigitYearStart(Date startDate) Sets the 100-year period 2-digit years will be interpreted as being in to begin on the date the user specifies. Parameters: startDate - During parsing, two digit years will be placed in the range startDate to startDate + 100 years. Since: 1.2 See Also: get2DigitYearStart() public Date get2DigitYearStart() Returns the beginning date of the 100-year period 2-digit years are interpreted as being within. Returns: the start of the 100-year period into which two digit years are parsed Since: 1.2 See Also: set2DigitYearStart(java.util.Date) public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition pos) Formats the given Date into a date/time string and appends the result to the given StringBuffer. Specified by: format in class DateFormat Parameters: date - the date-time value to be formatted into a date-time string. toAppendTo - where the new date-time text is to be appended. pos - the formatting position. On input: an alignment field, if desired. On output: the offsets of the alignment field. Returns: the formatted date-time string. Throws: NullPointerException - if the given date is null. public AttributedCharacterIterator formatToCharacterIterator(Object obj) Formats an Object producing an AttributedCharacterIterator. You can use the returned AttributedCharacterIterator to build the resulting String, as well as to determine information about the resulting String. Each attribute key of the AttributedCharacterIterator will be of type DateFormat.Field, with the corresponding attribute value being the same as the attribute key. Overrides: formatToCharacterIterator in class Format Parameters: obj - The object to format Returns: AttributedCharacterIterator describing the formatted value. Throws: NullPointerException - if obj is null. IllegalArgumentException - if the Format cannot format the given object, or if the Format's pattern string is invalid. Since: 1.4 public Date parse(String text, ParsePosition pos) Parses text from a string to produce a Date. The method attempts to parse text starting at the index given by pos. If parsing succeeds, then the index of pos is updated to the index after the last character used (parsing does not necessarily use all characters up to the end of the string), and the parsed date is returned. The updated pos can be used to indicate the starting point for the next call to this method. If an error occurs, then the index of pos is not changed, the error index of pos is set to the index of the character where the error occurred, and null is returned. This parsing operation uses the calendar to produce a Date. All of the calendar's date-time fields are cleared before parsing, and the calendar's default values of the date-time fields are used for any missing date-time information. For example, the year value of the parsed Date is 1970 with GregorianCalendar if no year value is given from the parsing operation. The TimeZone value may be overwritten, depending on the given pattern and the time zone value in text. Any TimeZone value that has previously been set by a call to setTimeZone may need to be restored for further operations. Specified by: parse in class DateFormat Parameters: text - A String, part of which should be parsed. pos - A ParsePosition object with index and error index information as described above. Returns: A Date parsed from the string. In case of error, returns null. Throws: NullPointerException - if text or pos is null. public String toPattern() Returns a pattern string describing this date format. Returns: a pattern string describing this date format. public String toLocalizedPattern() Returns a localized pattern string describing this date format. Returns: a localized pattern string describing this date format. public void applyPattern(String pattern) Applies the given pattern string to this date format. Parameters: pattern - the new date and time pattern for this date format Throws: NullPointerException - if the given pattern is null IllegalArgumentException - if the given pattern is invalid public void applyLocalizedPattern(String pattern) Applies the given localized pattern string to this date format. Parameters: pattern - a String to be mapped to the new date and time format pattern for this format Throws: NullPointerException - if the given pattern is null IllegalArgumentException - if the given pattern is invalid public void setDateFormatSymbols(DateFormatSymbols newFormatSymbols) Sets the date and time format symbols of this date format. Parameters: newFormatSymbols - the new date and time format symbols Throws: NullPointerException - if the given newFormatSymbols is null See Also: getDateFormatSymbols() public Object clone() Creates a copy of this SimpleDateFormat. This also clones the format's date format symbols. Overrides: clone in class DateFormat Returns: a clone of this SimpleDateFormat See Also: Cloneable public boolean equals(Object obj) Compares the given object with this SimpleDateFormat for equality. Overrides: equals in class DateFormat Parameters: obj - the reference object with which to compare. Returns: true if the given object is equal to this SimpleDateFormat See Also: Object.hashCode(), HashMap Submit a bug or feature For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples. Copyright © 1993, 2025, Oracle and/or its affiliates. All rights reserved. Use is subject to license terms. Also see the documentation redistribution policy. While working on Gozer, the static site generator responsible for building this website from a bunch of Markdown files, I got bit by a peculiarity around RFC822 formatted datetimes and Golang's constants for common datetime formats. I was working on adding support for RSS feeds, which is an XML feed that conforms to the RSS 2.0 specification. That specification states that all datetimes in RSS must conform to the Date and Time Specification of RFC 822. This format is important enough to have its own constant in Golang's time package so formatting a local time in that format should be as easy as: publishedAt.Format(time.RFC822) Except... W3C's feed validation service kept rejecting the RSS feed because of a "problematic RFC 822 date-time value". Huh. Is Golang outputting the wrong format or is W3C being weird? Who's wrong here? The datetime being rejected looked like this: Wed, 22 Nov 2023 16:33:55 CET While W3C's feed validation service shows the following as an example datetime which is valid: Wed, 02 Oct 2002 08:00:00 EST Can you spot the difference? Just looking at the format, I couldn't. Accepted timezone abbreviations for RFC 822 After taking a closer look at the RFC 822 specification their list of valid time zone abbreviations looked... lacking. zone = "UT" / "GMT" ; Universal Time ; North American : UT / "EST" / "EDT" ; Eastern: - 5/ - 4 / "CST" / "CDT" ; Central: - 6/ - 5 / "MST" / "MDT" ; Mountain: - 7/ - 6 / "PST" / "PDT" ; Pacific: - 8/ - 7 / 1ALPHA ; Military: Z = UT; ; A:-1; (J not used) ; M:-12; N:+1; Y:+12 / (("+" / "-") 4DIGIT) ; Local differential ; hours+min. (HHMM) It turns out that CET is not a valid time zone according to RFC 822. Only UTC, GMT and US time zone abbreviations are accepted... That lead me to checking ISO 8601, which doesn't accept any abbreviations because they can be ambiguous. For example, "CST" can mean China Standard Time (UTC+8), Cuba Standard Time (UTC−5), and (North American) Central Standard Time (UTC−6). It is therefore recommended to always specify the timezone in terms of its UTC offset. In this particular case the fix was to use time.RFC822Z as my format string, which specifies the timezone offset numerically. Today I learned! MIT/GNU Scheme provides a simple set of procedures for manipulating date and time information. There are four time representations, each of which serves a different purpose. Each representation may be converted to any of the others. The primary time representation, universal time, is an exact non-negative integer counting the number of seconds that have elapsed since midnight January 1, 1900 UTC. (UTC stands for Coordinated Universal Time, and is the modern name for Greenwich Mean Time.) This format is produced by get-universal-time and decoded-time->universal-time. The second representation, decoded time, is a record structure in which the time is broken down into components, such as month, minute, etc. Decoded time is always relative to a particular time zone, which is a component of the structure. This format is produced by global-decoded-time and local-decoded-time. The third representation, file time, is an exact non-negative integer that is larger for increasing time. Unlike universal time, this representation is operating-system dependent. This format is produced by all of the file-attribute procedures, for example file-modification-time and file-attributes. The fourth representation, the time string, is an external representation for time. This format is defined by RFC-822, Standard for the format of ARPA Internet text messages, with the modification that years are represented as four-digit numbers rather than two-digit numbers. This format is the standard format for Internet email and numerous other network protocols. Within this section, argument variables named universal-time, decoded-time, file-time, and time-string are respectively required to be of the corresponding format. Emails generated by our applications were displaying universal time instead of a user’s local time when viewed in Thunderbird. Our servers use universal time. The emails we send to our users include a date header with UTC. Other email clients automatically convert this to the user’s local time. I compared the date header in our messages to date headers in other messages in my inbox. I found that there were a variety of formats and wasn’t sure which one was correct, so I turned to the RFCs. I found RFC 4021 first, which referred to RFC 2822 and RFC 822. The proper “date-time” syntax was originally defined in RFC 822 section 5.1. and later clarified in RFC 2822 section 3.3. Even the clarifications in RFC 2822 allowed several different formats, so I analyzed 132,037 date headers on one of our mail servers, hoping to determine if a specific format was most common. I found that the following format is by far the most common: Date: Tue, 18 Nov 2014 15:57:11 +0000 The day of month and hour of day are both two digits. The time zone is a 4-digit offset prefixed with either a “+” or “-”. In the example above, the server is set to GMT or UTC, so the offset is “+0000”. Be aware that “+0000” and “-0000” are not handled the same! RFC 2822 section 3.3 says that offset “-0000” should be treated as an unknown timezone. The following PHP code will output the format above: echo "Date: " . date("D, d M Y H:i:s O"); It also seems acceptable to place the time zone or a comment in parenthesis after the date. Here are a few examples: Date: Fri, 03 Dec 2010 16:02:30 -0600 (CST) Date: Fri, 9 Sep 2005 16:38:47 -0400 (added by postmaster@attrh1i.attrh.att.com) I have attached the data file that contains 130,606 date headers (after removing 2k of mangled records that included other email contents) in case you would like to perform additional analysis. smtp-header-dates-20141117.txt RFC # 822 Obsoletes: RFC #733 (NIC #41952) Revised by David H. CrockerDept. of Electrical EngineeringUniversity of Delaware, Newark, DE 19711Network: DCrocker @ UDel-Relay Partial Hypertext conversion by Tim Berners-Lee/CERN PREFACE ..... ii 1. INTRODUCTION ..... 1 1.1. Scope ..... 1 1.2. Communication Framework ..... 2 2. NOTATIONAL CONVENTIONS ..... 3 3. LEXICAL ANALYSIS OF MESSAGES 3.1. General Description 3.2. Header Field Definitions 3.3. Lexical Tokens 3.4. Clarifications 4. MESSAGE SPECIFICATION ..... 17 4.1. Syntax ..... 17 4.2. Forwarding ..... 19 4.3. Trace Fields ..... 20 4.4. Originator Fields ..... 21 4.5. Receiver Fields ..... 23 4.6. Reference Fields ..... 23 4.7. Other Fields ..... 24 5. DATE AND TIME SPECIFICATION ..... 26 5.1. Syntax ..... 26 5.2. Semantics ..... 26 6. ADDRESS SPECIFICATION ..... 27 6.1. Syntax ..... 27 6.2. Semantics ..... 27 6.3. Reserved Address ..... 33 7. BIBLIOGRAPHY ..... 34 APPENDIX A. EXAMPLES ..... 36 B. SIMPLE FIELD PARSING ..... 40 C. DIFFERENCES FROM RFC #733 ..... 41 D. ALPHABETICAL LISTING OF SYNTAX RULES ..... 44 By 1977, the Arpanet employed several informal standards for the text messages (mail) sent among its host computers. It was felt necessary to codify these practices and provide for those features that seemed imminent. The result of that effort was Request for Comments (RFC) #733, "Standard for the Format of ARPA Network Text Message", by Crocker, Vittal, Pogran, and Henderson. The specification attempted to avoid major changes in existing software, while permitting several new features. This document revises the specifications in RFC #733, in order to serve the needs of the larger and more complex ARPA Internet. Some of RFC #733's features failed to gain adequate acceptance. In order to simplify the standard and the software that follows it, these features have been removed. A different addressing scheme is used, to handle the case of inter-network mail; and the concept of re-transmission has been introduced. This specification is intended for use in the ARPA Internet. However, an attempt has been made to free it of any dependence on that environment, so that it can be applied to other network text message systems. The specification of RFC #733 took place over the course of one year, using the ARPANET mail environment, itself, to provide an on-going forum for discussing the capabilities to be included. More than twenty individuals, from across the country, participated in the original discussion. The development of this revised specification has, similarly, utilized network mail-based group discussion. Both specification efforts greatly benefited from the comments and ideas of the participants. The syntax of the standard, in RFC #733, was originally specified in the Backus-Naur Form (BNF) meta-language. Ken L. Harrenstien, of SRI International, was responsible for recoding the BNF into an augmented BNF that makes the representation smaller and easier to understand. 1. INTRODUCTION This standard specifies a syntax for text messages that are sent among computer users, within the framework of "electronic mail". The standard supersedes the one specified in ARPANET Request for Comments #733, "Standard for the Format of ARPA Net- work Text Messages". In this context, messages are viewed as having an envelope and contents. The envelope contains whatever information is needed to accomplish transmission and delivery. The contents compose the object to be delivered to the recipient. This stan- dard applies only to the format and some of the semantics of mes- sage contents. It contains no specification of the information in the envelope. However, some message systems may use information from the contents to create the envelope. It is intended that this stan- dard facilitate the acquisition of such information by programs. Some message systems may store messages in formats that differ from the one specified in this standard. This specifica- tion is intended strictly as a definition of what message content format is to be passed BETWEEN hosts. Note: This standard is NOT intended to dictate the internal for- mats used by sites, the specific message system features that they are expected to support, or any of the charac- teristics of user interface programs that create or read messages. A distinction should be made between what the specification REQUIRES and what it ALLOWS. Messages can be made complex and rich with formally-structured components of information or can be kept small and simple, with a minimum of such information. Also, the standard simplifies the interpretation of differing visual formats in messages; only the visual aspect of a message is affected and not the interpretation of information within it. Implementors may choose to retain such visual distinctions. The formal definition is divided into four levels. The bot- tom level describes the meta-notation used in this document. The second level describes basic lexical analyzers that feed tokens to higher-level parsers. Next is an overall specification for messages; it permits distinguishing individual fields. Finally, there is definition of the contents of several structured fields. Messages consist of lines of text. No special provisions are made for encoding drawings, facsimile, speech, or structured text. No significant consideration has been given to questions of data compression or to transmission and storage efficiency, and the standard tends to be free with the number of bits con- sumed. For example, field names are specified as free text, rather than special terse codes. A general "memo" framework is used. That is, a message con- sists of some information in a rigid format, followed by the main part of the message, with a format that is not specified in this document. The syntax of several fields of the rigidly-formated ("headers") section is defined in this specification; some of these fields must be included in all messages. The syntax that distinguishes between header fields is specified separately from the internal syntax for particular fields. This separation is intended to allow simple parsers to operate on the general structure of messages, without concern for the detailed structure of individual header fields. Appendix B is provided to facilitate construction of these parsers. In addition to the fields specified in this document, it is expected that other fields will gain common use. As necessary, the specifications for these "extension-fields" will be published through the same mechanism used to publish this document. Users may also wish to extend the set of fields that they use privately. Such "user-defined fields" are permitted. The framework severely constrains document tone and appear- ance and is primarily useful for most intra-organization communi- cations and well-structured inter-organization communication. It also can be used for some types of inter-process communica- tion, such as simple file transfer and remote job entry. A more robust framework might allow for multi-font, multi-color, multi- dimension encoding of information. A less robust one, as is present in most single-machine message systems, would more severely constrain the ability to add fields and the decision to include specific fields. In contrast with paper-based communica- tion, it is interesting to note that the RECEIVER of a message can exercise an extraordinary amount of control over the message's appearance. The amount of actual control available to message receivers is contingent upon the capabilities of their individual message systems. This specification uses an augmented Backus-Naur Form (BNF) notation. The differences from standard BNF involve naming rules and indicating repetition and "local" alternatives. Angle brackets ("") are not used, in general. The name of a rule is simply the name itself, rather than "". Quotation-marks enclose literal text (which may be upper and/or lower case). Certain basic rules are in uppercase, such as SPACE, TAB, CRLF, DIGIT, ALPHA, etc. Angle brackets are used in rule definitions, and in the rest of this document, whenever their presence will facilitate discerning the use of rule names. Elements separated by slash ("/") are alternatives. There- fore "foo / bar" will accept foo or bar. 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". The character "*" preceding an element indicates repetition. The full form is: *element indicating at least and at most 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. Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)". "(element)" is equivalent to "*(element)"; that is, exactly occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters. A construct "#" is defined, similar to "*", as follows: #element indicating at least and at most elements, each separated by one or more commas (","). This makes the usual form of lists very easy; a rule such as '(element *("," 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 ele- ment 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. 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. A message consists of header fields and, optionally, a body. The body is simply a sequence of lines containing ASCII characters. It is separated from the headers by a null line (i.e., a line with nothing preceding the CRLF). Each header field can be viewed as a single, logical line of ASCII characters, comprising a field-name and a field-body. For convenience, the field-body portion of this conceptual entity can be split into a multiple-line representation; this is called "folding". The general rule is that wherever there may be linear-white-space (NOT simply LWSP-chars), a CRLF immediately followed by AT LEAST one LWSP-char may instead be inserted. Thus, the single line To: "Joe & J. Harvey" , JJV @ BBN can be represented as: To: "Joe & J. Harvey" , JJV@BBN and To: "Joe & J. Harvey" , JJV @BBN and To: "Joe & J. Harvey" , JJV @ BBN The process of moving from this folded multiple-line representation of a header field to its single line representation is called "unfolding". Unfolding is accomplished by regarding CRLF immediately followed by a LWSP-char as equivalent to the LWSP-char. While the standard permits folding wherever linear-white-space is permitted, it is recommended that structured fields, such as those containing addresses, limit folding to higher-level syntactic breaks. For address fields, it is recommended that such folding occur between addresses, after the separating comma. Once a field has been unfolded, it may be viewed as being composed of a field-name followed by a colon (":"), followed by a field-body, and terminated by a carriage-return/line-feed. The field-name must be composed of printable ASCII characters (i.e., characters that have values between 33. and 126., decimal, except colon). The field-body may be composed of any ASCII characters, except CR or LF. (While CR and/or LF may be present in the actual text, they are removed by the action of unfolding the field.) Certain field-bodies of headers may be interpreted according to an internal syntax that some systems may wish to parse. These fields are called "structured fields". Examples include fields containing dates and addresses. Other fields, such as "Subject" and "Comments", are regarded simply as strings of text. Note:Any field which has a field-body that is defined as other than simply is to be treated as a structured field. Field-names, unstructured field bodies and structured field bodies each are scanned by their own, independent "lexical" analyzers. 3.1.3. UNSTRUCTURED FIELD BODIESFor some fields, such as "Subject" and "Comments", no structuring is assumed, and they are treated simply as s, as in the message body. Rules of folding apply to these fields, so that such field bodies which occupy several lines must therefore have the second and successive lines indented by at least one LWSP-char. 3.1.4. STRUCTURED FIELD BODIESTo aid in the creation and reading of structured fields, the free insertion of linear-white-space (which permits folding by inclusion of CRLFs) is allowed between lexical tokens. Rather than obscuring the syntax specifications for these structured fields with explicit syntax for this linear-white-space, the existence of another "lexical" analyzer is assumed. This analyzer does not apply for unstructured field bodies that are simply strings of text, as described above. The analyzer provides an interpretation of the unfolded text composing the body of the field as a sequence of lexical symbols. These symbols are: individual special characters quoted-strings domain-literals comments atoms The first four of these symbols are self-delimiting. Atoms are not; they are delimited by the self-delimiting symbols and by linear-white-space. For the purposes of regenerating sequences of atoms and quoted-strings, exactly one SPACE is assumed to exist, and should be used, between them. (Also, in the "Clarifications" section on "White Space", below, note the rules about treatment of multiple contiguous LWSP-chars.) So, for example, the folded body of an address field ":sysmail"@ Some-Group. Some-Org, Muhammed.(I am the greatest) Ali @(the)Vegas.WBA is analyzed into the following lexical symbols and types: :sysmail quoted string @ special Some-Group atom . special Some-Org atom , special Muhammed atom . special (I am the greatest) comment Ali atom @ atom (the) comment Vegas atom . special WBA atom The canonical representations for the data in these addresses are the following strings: ":sysmail"@Some-Group.Some-Org and Muhammed.Ali@Vegas.WBA Note:For purposes of display, and when passing such structured information to other systems, such as mail protocol services, there must be NO linear-white-space between s that are separated by period (".") or at-sign ("@") and exactly one SPACE between all other s. Also, headers should be in a folded form. 3.2. HEADER FIELD DEFINITIONSThese rules show a field meta-syntax, without regard for the particular type or internal syntax. Their purpose is to permit detection of fields; also, they present to higher-level parsers an image of each field as fitting on one line. field = field-name ":" [field-body] CRLF field-name = 1* field-body = field-body-contents [CRLF LWSP-char field-body] field-body-contents = 3.3. LEXICAL TOKENSThe following rules are used to define an underlying lexical analyzer, which feeds tokens to higher level parsers. See the ANSI references, in the Bibliography. ; (Octal, Decimal.) CHAR = ; (0-177, 0.-127.) ALPHA = ; (101-132, 65.- 90.) ; (141-172, 97.-122.) DIGIT = ; (60- 71, 48.- 57.) CTL = ; (177, 127.) CR = ; (15, 13.) LF = ; (12, 10.) SPACE = ; (40, 32.) HTAB = ; (11, 9.) ; string, to use / "." / "[" / "]" ; within a word. delimiters = specials / linear-white-space / comment text = atoms, specials, CR & bare LF, but NOT ; comments and including CRLF> ; quoted-strings are ; NOT recognized. atom = 1* quoted-string = ; Regular qtext or ; quoted chars. qtext = , ; => may be folded "\" & CR, and including linear-white-space> quoted-pair = "\" CHAR ; may quote any char quoted-string = ; Regular qtext or ; quoted chars. received = "Received" ":" ; one per relay ["from" domain] ; sending host ["by" domain] ; receiving host ["via" atom] ; physical path *("with" atom) ; link/mail protocol ["id" msg-id] ; receiver msg id ["for" addr-spec] ; initial form ";" date-time ; time received resent = resent-authentic ["Resent-Reply-To" ":" 1#address]) resent-authentic = = "Resent-From" ":" mailbox / ("Resent-Sender" ":" mailbox "Resent-From" ":" 1#mailbox) resent-date = "Resent-Date" ":" date-time return = "Return-path" ":" route-addr ; return address route = 1#("@" domain) ":" ; path-relative route-addr = "" source = [trace] ; net traversals originator ; original mail [resent] ; forwarded SPACE = ; (40, 32.) specials = "(" / ")" / "" / "@" ; Must be in quoted- / "," / ";" / ":" / "\" / = ; (42, 34.)