http://www.w3.org/TR/hdml20-3.html Go
59 captures
6 Jun 1997 - 1 Oct 2023
MAY JUN MAY
1996 06 1997 2000
About this capture

Preface

Introduction
Handheld Devices
Physical characteristics

The Handheld Device Markup Language (HDML) is a simple language used to create hypertext-like content for small display, handheld devices. This specification defines HDML version 2.0.

Introduction

The World Wide Web provides a robust, flexible, and ubiquitous model for information access. The adoption of the WWW as the preferred means of disseminating and accessing information from desktop PCs and workstations has created a demand for access to the same information for other devices. These devices or "alternate platforms" range from voice- and fax-based user agents to low-cost Network Computers to handheld devices such as mobile phones and PDAs.

While the web's infrastructure and protocols fully support these alternate platforms, HTML itself does not. In particular the navigation and display models inherent to HTML collapse when applied to a typical handheld device. However, combining the use of standard web protocols and infrastructure (URLs, HTTP, SSL plus CGI, Perl, commercial web servers) with an alternate but complementary markup language, allows handheld devices to function as full-fledged web clients.

Handheld Devices

While there are many types, styles, and classes of handheld devices, this specification is useful for a significantly larger class of devices with similar physical characteristics. Those characteristics include:

-
  - small display size (always)
  - limited input capabilities (often)
  - limited bandwidth both (often)
  - limited resources such as memory, processing power, permanent storage (often)

This specification will use the data-ready mobile phone as an example of the typical handheld device. This specification will also continue to use the term "handheld," recognizing that the term is not inclusive of all devices which would benefit from HDML.

Physical characteristics

Handheld devices are characterized primarily by a limited display size. A typical display is capable of displaying 4-10 lines of text 12-20 characters wide and may be graphical (bitmapped) or text-only. PDA-style displays are not necessarily included in this handheld device category, although HDML will be useful on those devices as well.

Handheld devices may or may not have a full keyboard and may or may not have a pointing/selection device. As an example, the data-ready mobile phone has only:

-
  - the keys normally found on a telephone (0-9, *, #, with alphabet letters marked on 2-9)
  - cursor/arrow keys (often just up and down or left and right)
  - a number of dedicated function keys (SEND, END, etc.)
  - one or more "soft keys" with programmable labels

Alphanumeric pagers have even fewer keys. Item selection is accomplished through numbered lists or using cursor keys to

highlight a choice and then request an action on that item. Full 2-D cursor control through pointing devices such as touch pads, touch screens, roller balls, are rare. A full QWERTY keyboard is also rare. Input is severely constrained when compared to a typical PC or PDA.

Network bandwidth is usually low due to limitations of the network technology or simple economics. The same goes for other resources: memory, processing power, even battery life. All in the name of economics. While some devices do have large amounts of memory or processing power, these devices are the exception. Mass market and consumer-targeted devices will continue to have these constraints for many years.

IPR2025-00857
Apple EX1014 Page 2

http://www.w3.org/TR/hdml20-4.html Go
72 captures
6 Jun 1997 - 8 Dec 2023
MAY JUN MAY
1996 06 1997 2000
About this capture

Chapter 1 : Overview

Cards
Decks
Actions
Activities
History-based navigation
Variables
Setting Variables
Accessing Variables
Passing Variables
Returning Variables from Sub-activities
Security Features
Deck Access Control
SendReferer

HDML is more than just a language. Like many languages, it requires a run-time environment to make it useful. This document not only specifies the language at a lexical and syntactic level but also specifies the run-time environment at a semantic level.

The element that provides the run-time environment for HDML is referred to as the *user agent*.

Cards

The fundamental building block of HDML content is the *card*. The user agent displays and allows the user to interact with cards of information. Logically, a user navigates through a series of HDML cards, reviews the contents of each, enters requested information, makes choices, and moves on to another or returns to a previously visited card.

Cards come in one of four forms:

-
  - *Nodisplay*
  - *Display*
  - *Choice*
  - *Entry*

Display, choice, and entry cards contain text and/or references to images that are displayed to the user. Choice cards allow the user to pick from a list of available options, and entry cards allow the user to enter text.

While it is expected that cards contain short pieces of information, they might contain more information than can be displayed in one screen full. The user agent will provide a mechanism for the user to view the entire contents of the card. An example of this would be a user-interface that allows scrolling through the information.

Decks

HDML cards are grouped together into *decks*. An HDML deck is similar to an HTML page in that it is identified by URL [RFC1738] and is the unit of content requested from a server and cached by the user agent.

Individual cards within a deck are identified by name and can be addressed by URL fragment. For example, the URL of "Card1" in "http://www.foo.com/MyDeck.hdml" is:

```
http://www.foo.com/MyDeck.hdml#Card1
```

The first card in a deck has the same URL as the deck itself. It does not need to be named unless other cards within that deck use it as a destination.

A user agent may store HDML decks in a local cache. If it does so, the user agent shall provide a means to flush decks from its cache by URL. HDML decks can specify their time-to-live (TTL) in the user agent's cache.

Decks can also specify default actions for the cards it contains (see "Actions" below).

Actions

After completing interactions with a card, a user may select one of the following *actions:*

ACCEPT	**Complete text entry, move forward to the next card, start or complete an activity; this is the primary action for moving forward**
PREV	Move back to the previous card
SOFT1-SOFT8	Application defined
HELP	Display a help card to the user

The actions defined in HDML are abstract and are supported in the user interface in a user agent specific manner. For example, a data-ready mobile phone with two soft keys may map the left soft key to the ACCEPT action, the right soft key to the SOFT1 action, and dedicated keys to the PREV and HELP actions.

Each action, when selected, performs a task. For example, the GO task displays another card (either in the current deck or a different deck), and (in the case of a data-ready mobile phone) the CALL task starts a voice call to a particular phone number.

The mapping of action to task can be specified in the following places (in order of precedence):

-
    - The selected choice entry (in a choice card) is a variant of the ACCEPT action
    - The current card can contain a list of actions
    - The deck can contain a list of actions

If not specified, actions have the following default behavior

ACCEPT	Display the previous card in the history
PREV	Display the previous card in the history or cancel the current activity if there is no history (see "History-based navigation")
SOFTx	Do nothing
HELP	Display a built-in help card

The next card the user agent will process in an interaction is referred to as a destination. An HDML destination is similar to a web URL with some extended semantics as described in "Destinations".

Activities

The HDML navigation and state model is structured around activities. An activity can be thought of as a group of related cards that perform one logical function. These cards may span one or more decks. Navigation among cards within an activity is non-destructive, that is, UI state is preserved if you move back to a previous card and forward again. When you exit an activity, all card-related information is lost.

Activities are managed on the *activity stack*. Entering an activity pushes it on the stack, and exiting an activity pops it off the stack. In this way, activities are similar to subroutines. An activity is entered using the GOSUB task and is exited using either the RETURN or CANCEL tasks (see "Tasks").

The activity stack is used for:

-
  - Grouping user-interface activities
    The activity stack can be used to group a number of cards together into a single, complex user interaction, for example, entering data into multiple cards. From any of the cards, the user can return from or cancel out of the entire activity.
  - History-based navigation
    Each activity maintains a history of the cards displayed to simplify implementing navigation.
  - State management
    Variables are associated with and scoped by activities (see "Variables").

History-based navigation

In each activity, the user agent maintains a history of the cards displayed to the user. As each card is accessed, it is added to the card history. The user agent allows the user to easily navigate back to the previous card in the history.

A card can override the history by defining a PREV action. Note that accessing a card that specifies a PREV action has the effect of invalidating the current activity's history. It is also possible for other actions to navigate to the previous card in the history by invoking the PREV task.

Note that the PREV *action* and PREV *task* are separate and distinct concepts. There are situations where an application might want to override the default PREV action, such as during navigation of a long list of cards. If the application lets the user search and display a card in the middle of the lists, it may overload the PREV action so that it always takes you to the previous card in the list regardless of whether it is in the history or not. Conversely, there are situations where an application may want the ACCEPT action to take the user to the previous card in the history. To accomplish this, the ACCEPT action is bound to the PREV task.

Variables

Variables are named pieces of information that can be substituted into cards as they are displayed and otherwise processed. Variables are scoped to the current activity.

Setting Variables

There are a number of ways to set variables. Choice and entry cards set variables. The KEY option of the card specifies the name of the variable to set. For entry cards, the value to whatever the user has entered, and for choice cards, the choice entries specify the value in their VALUE options. Variables can be set when navigating from one card to another. Variables can also be set when a sub-activity returns.

Accessing Variables

The values of variables can be substituted into both the displayed text of a deck and certain option values in HDML elements. The substitution of variable values happens at run-time, in the user agent.

Passing Variables

Variables can be set when navigating from one card to the next. There are two ways to pass variables. GO and GOSUB tasks use the VARS option to specify the variables to pass. In the example that follows, the ACCEPT action causes the user agent to go to "card2", and in the process, to set the variable "name" to "John".

```
<ACTION TYPE=ACCEPT TASK=GO DEST="#card2" VARS="name=John">
```

Similarly, variables can be passed into a sub-activity. In the following example, a new sub-activity is created with the one initial variable, "name", whose value is "Jane". The first card displayed in the sub-activity is the first card in the deck "sub.hdml".

```
<ACTION TYPE=ACCEPT TASK=GOSUB DEST="sub.hdml" VARS="name=Jane">
```

Returning Variables from Sub-activities

A card can make a call to a sub-activity to have it return new values for variables. For example, a sub-activity may allow the user to pick an email address out of an address book and then return it to the calling activity. Using a deck in this way is similar to calling a subroutine in a computer programming language.

A list of values to be returned from an activity is specified in the RETVALS option of the RETURN task. The names of the variables to assign the values are specified in the RECEIVE option of the calling action. For example:

```
<HDML version=2.0>
    <DISPLAY>
    <ACTION TYPE=ACCEPT TASK=GOSUB DEST="#subact"
     RECEIVE="var1;var2">
    var1 = $(var1)<BR>
    var2 = $(var2)
    </DISPLAY>

    <DISPLAY NAME=subact>
    <ACTION TYPE=ACCEPT TASK=RETURN RETVALS="abc;xyz">
    Return abc;xyz
    </DISPLAY>
</HDML>
```

In the above example, the sub-activity returns two values, "abc", and "xyz". The calling action assigns these values to the variables "var1" and "var2" respectively. Note that the values are assigned based on their position in the retvals list. A calling action can ignore a return value simply by not specifying a variable name for that position. Similarly, if a variable name is specified for a position that has no value, the variable will be cleared, i.e. set to a zero-length string.

Security Features

HDML provides the following security features.

Deck Access Control

The existence of variable state in the language and runtime introduces some security issues that do not exist in other markup languages such as HTML. In particular, certain variable state may be considered private by the user. While he or she may be willing to send a credit card number to a secure service, an insecure and nefarious service should not be able to retrieve that number from the user agent by any means.

Also, HDML applications often use variables to consolidate redundant information to parameterize decks. The basic reliability of an application could potentially be compromised by a card or deck being invoked with invalid or malicious variables.

The user agent performs deck level access control when navigating from one card to the next. A deck can be declared as *public*, which means it can be accessed from any other deck. Also, decks that are not public can specify which decks can access them by URL. This allows an application to protect decks from the attacks. See "The HDML Element" for more details.

SendReferer

Typically, when an HDML application takes action on behalf of the user, it does so as a side-effect of the user agent accessing a URL. For example, requesting a URL from an HTTP server can cause a CGI program to run. This CGI program can do some things, and then return an HDML deck to display the results. Thus, some URLs have side-effects.

If an application wants to protect a URL that has destructive or other side-effects, it should allow the URL to be invoked from certain other URLs, that is, HDML decks that were generated by the application itself.

The HDML user agent should provide the capability to communicate the referring URL to the application. How this is communicated depends on the client-server protocol. In the case of HTTP [RFC2068], the user agent will specify the referring URL in the "Referer" request header.

In some network configurations, it is prohibitively expensive or infeasible to send the referring URL in every request to a server. To accommodate these networks, HDML allows the application to specify whether to send the referring URL in the request or not. This strikes the balance between security for URLs with side effects and network costs.

IPR2025-00857
Apple EX1014 Page 6

http://www.w3.org/TR/hdml20-5.html Go
66 captures
6 Jun 1997 - 10 Dec 2023
MAY JUN AUG
1996 06 1997 2000
About this capture

Chapter 2 : Language Syntax

Lexical Structure
Options

The structure of HDML is described by an abstract syntax using an enhanced BNF:

a ::= b	the element a is defined as b
a ::= b ::= c	the element a is defined as b or c
b c	element b is followed by element c
a | b | c	element a or element b or element c
{ a }	the element a is optional
{ a } *	the element a may appear zero or more times in a row
(a) +	the element a may appear one or more times in a row
abc	the characters abc literally
ol(a)	an option list with zero or more options of the element a, "Options"

Lexical Structure

HDML 2.0 is nominally an ISO-Latin-1(ISO 8859-1) language. Syntactic elements (delimiters, keywords, option names, and the like) are constrained to US-ASCII printable characters and whitespace so that other 8-bit character encodings can be used as long as:

-
  - The character encoding is a superset of US-ASCII
  - The character encoding is communicated to the user agent
  - The characters in the document are printable or white-space

If the character encoding is not specified, ISO-Latin-1 may be assumed.

With the exceptions described below, white space is used as a delimiter, and is otherwise ignored. HDML white space is defined as US-ASCII newline (10), carriage return (13), space (32) and tab (9).

HDML also supports comments. Comments are treated as white space, and have the following syntax:

-
```
comment ::= <-- text -->
        ::= <!-- text -->
```

In order to disambiguate between syntactic elements and the contents of the decks and cards, HDML uses SGML character and entity references to allow you to quote special characters. The following table lists the available references.

Character/Entity Reference	Character
>	>
<	<
&	&

&dol;	$
	non-breaking space
&# hex hex ;	any one-byte character

Character and entity references are not allowed in tag, option, and variable names, but they are allowed in option values and formatted text.

Options

Many of the syntactic elements of HDML have option lists associated with them. Options refine the operation of the elements they are part of. Options that are not defined for an element are ignored.

```
ol(valid-option) ::= { valid-option }*

     option-list ::= ol(option)

          option ::= key = value

             key ::= identifier

           value ::= plain-text

                 ::= " { text }* "

 value-with-vars ::= variable | plain-text

                 ::= " { variable | text }* "
```

An option list contains zero or more options. Each option is separated by blank-space and optionally followed by blank-space. In the syntax diagrams, option lists are shown as: ol(*valid-option*) where *valid-option* is replaced with an element that defines the possible options in this context. ol is a generic syntactic description of option-lists.

Each option is a key and a value. Options be given in any order within the list of options. The key is always an alpha-numeric name that is case insensitive. If the value of an option does not contain whitespace, it may appear directly after the equals sign, otherwise, the value must be quoted. Values are composed of a sequence of text and, if allowed, variable substitutions. In quotes, blank-space (including newlines) is treated literally and is considered part of the value.

In the syntax diagrams, the possible values for various options are specified without quotes. However, quotes are always acceptable around an option value.

Many options have a more restricted set of possible values than represented by the above syntax. See the individual options for details.

IPR2025-00857
Apple EX1014 Page 9

http://www.w3.org/TR/hdml20-6.html Go
83 captures
6 Jun 1997 - 17 Jun 2024
MAY JUN AUG
1996 06 1997 2000
About this capture

Chapter 3 : Language Elements

The HDML Element
VERSION
TTL
PUBLIC
ACCESSPATH and ACCESSDOMAIN
MARKABLE
Card Elements
NAME
TITLE
BOOKMARK
The DISPLAY Card Element
The NODISPLAY Card Element
The CHOICE Card Element
METHOD
KEY and DEFAULT
IKEY and IDEFAULT
Choice Entries (The CE Element)
VALUE
The ENTRY Card Element
KEY and DEFAULT
FORMAT
NOECHO
EMPTYOK
Formatted Text
Line Formatting
Alignment Formatting
Text Formatting
Tab Formatting
Images
SRC
NAME
ALT
The ACTION Element
TYPE
LABEL
IMAGE
Destinations
Tasks
TASK
DEST
VARS
RECEIVE
RETVALS
NEXT
CANCEL
SENDREFERER
FRIEND
CLEAR
NUMBER
Variables
Setting Variables

This section describes the elements of the language and their semantics.

Not all of the BNF statements can be found in this section. For a complete listing of the HDML BNF, see "HDML 2.0 BNF Summary."

The HDML Element

```
          deck ::= deck-header { action }* { card }+ deck-footer
   deck-header ::= <HDML o-version ol(deck-options) >
  deck-options ::= o-ttl | o-deckseq | o-public |
                   o-accessdomain | o-accesspath | o-markable
   deck-footer ::= </HDML>
```

A deck starts with the <HDML> tag and ends with the </HDML> tag. It contains an optional list of default actions and one or more cards. There must be at least one card. Actions specified at the deck level are used as default actions by the whole deck. Individual cards may override deck actions as needed. See "The ACTION Element" for syntax and full description.

VERSION

```
o-version := version = value
```

The version of the deck indicates the particular version of the HDML language that is used in the rest of the deck. It is of the form m.n where m is the major version number and n is the minor version number. This specification describes HDML version 2.0.

TTL

```
o-ttl ::= ttl = {numeric}+
```

Decks can be cached by the user agent for a period of time. The TTL option indicates the number of seconds that the deck can be cached from time of reception. If no TTL is given, the deck will be cached for an indeterminate period of time. If the value of TTL is 0 (zero) then the deck is cached only in the context of the current activity. See "Deck Time-to-Live" for more details on the behavior of expired decks.

PUBLIC

```
o-public ::= public = boolean-value
```

The PUBLIC option indicates whether deck access control has been enabled for this deck. If PUBLIC=TRUE, cards in any deck can access the deck. If PUBLIC=FALSE, then the ACCESSDOMAIN and ACCESSPATH options are enabled and indicate which decks can access the deck. The default value of PUBLIC=FALSE.

ACCESSPATH and ACCESSDOMAIN

```
o-accessdomain ::= accessdomain = url
  o-accesspath ::= accesspath = path
```

The ACCESSDOMAIN and ACCESSPATH options specify which decks may access the deck. When one card goes to another card, access control is performed to determine whether that destination is allowed. The access control options protect cards that, for example, return sensitive information.

If a deck has an ACCESSDOMAIN and/or ACCESSPATH option, the referring deck's URL must match the values of the options. Matching is done as follows: the access domain is suffix-matched against the domain name portion of the referring URL, and the access path is prefix matched against the path portion of the referring URL.

The ACCESSDOMAIN option defaults to the current deck's domain. The ACCESSPATH option defaults to "/".

To simplify the development of applications which may not know the absolute path to the current deck, the ACCESSPATH option will accept relative URLs. The user agent will convert the relative path to an absolute path and then perform prefix matching against the ACCESSPATH option.

Given the following access control options:

```
ACCESSDOMAIN="uplanet.com"
ACCESSPATH="/cbb"
```

The following referring URLs would be allowed to go to the deck:

```
http://red.uplanet.com/cbb/stocks.cgi
https://yellow.uplanet.com/cbb/bonds.cgi
http://uplanet.com/cbb/demos/alpha/packages.cgi?x=123&y=456
```

The following referring URLs would not be allowed to go to the deck:

```
http://up.dbc.com/cbb
http://up.uplanet.com/internal/foo.hdml
```

MARKABLE

```
o-markable ::= markable = boolean-value
```

A user agent can save the location and state of the currently viewed card by *bookmarking* it. The MARKABLE option specifies whether the cards in this deck (or this card if specified as a card option) can be bookmarked or not. Typically, bookmarking should be disabled for decks that are returned as a result of a transaction or other URL that has side-effects. If bookmarking is enabled, the transaction or side-effect could be repeated when the URL is invoked again by the user agent. Specifying the MARKABLE=TRUE enables bookmarking. The default is FALSE.

MARKABLE=TRUE on a deck implies PUBLIC=TRUE on the deck.

Card Elements

```
card ::= display-card | nodisplay-card | choice-card | entry-card
```

A card is one of four types: DISPLAY, NODISPLAY, CHOICE, and ENTRY. Each type is described in a section below.

```
card-options ::= o-name | o-title | o-bookmark | o-markable
```

All card types can have the above options.

NAME

```
o-name ::= name = identifier
```

The optional NAME option gives a name to the card. If a card has a name, then it can be referred to as a fragment in a destination.

TITLE

```
o-title ::= title = value-with-vars
```

The TITLE option specifies the title of the card. If no TITLE option is specified, the first text line of the card is used as the title. The title is used in the following ways:

- Suggested bookmark name (when you mark the card)
- Text entry prompt (in entry cards)

BOOKMARK

```
o-bookmark ::= bookmark = destination
```

The BOOKMARK option specifies the URL to use when bookmarking the card. The default is the URL of the current card. This option is used if you want to force the bookmark to go to another card (like a NODISPLAY card that sets up variables) instead of the current card.

The DISPLAY Card Element

```
    display-card ::= display-header display-content display-footer
  display-header ::= <DISPLAY ol(display-options) >
 display-options ::= card-options
 display-content ::= { action }* formatted-text
  display-footer ::= </DISPLAY>
```

DISPLAY cards give information for the user to read. See "Formatted Text" below for a full description of the format of information that can be displayed. DISPLAY cards may also contain actions. These override any actions specified at the deck level.

```
<DISPLAY>
<ACTION TYPE=ACCEPT LABEL="Done">
    You just won the lottery!
</DISPLAY>
```

The NODISPLAY Card Element

```
     nodisplay-card ::= nodisplay-header nodisplay-content nodisplay-footer
   nodisplay-header ::= <NODISPLAY ol(nodisplay-options) >
  nodisplay-options ::= card-options
  nodisplay-content ::= { action }*
   nodisplay-footer ::= </NODISPLAY>
```

NODISPLAY cards do not give information for the user to read. Rather, they immediately execute the card's ACCEPT or PREV action. All other actions are ignored. The following table lists the actions taken depending on the manner in which the NODISPLAY card was invoked.

Invoking Task	Action-Taken
TASK=GO	ACCEPT
TASK=RETURN or TASK=CANCEL and DEST=*this card*	ACCEPT
TASK=RETURN and caller's NEXT=*this card*	ACCEPT
TASK=CANCEL and caller's CANCEL=*this card*	ACCEPT
TASK=RETURN	PREV
TASK=PREV	PREV
TASK=CANCEL	PREV

The CHOICE Card Element

```
    choice-card ::= choice-header display-content { entries } choice-footer
  choice-header ::= <CHOICE ol(choice-options) >
 choice-options ::= card-options | o-method | o-key | o-ikey | o-default |
                    o-idefault
        entries ::= { choice-entry }+
  choice-footer ::= </CHOICE>
```

CHOICE cards let users pick from a list of choices. The initial display content is shown to the user, followed by the choices. Each choice can have one line of formatted text (which may be wrapped by the user agent if too long).

```
<CHOICE KEY=name>
    Choose a name:
<CE VALUE=John>John Doe
<CE VALUE=Jane>Jane Dove
</CHOICE>
```

METHOD

```
      o-method ::= method = method-type
   method-type ::= number | alpha
```

How the choices are displayed and chosen is based on the METHOD option. This option is a hint only, and can be disregarded by the user agent. The NUMBER method indicates that the user agent should number the list and should allow quick selection of a choice entry by number, if possible. The ALPHA method indicates that the user agent should not number the list and should allow

quick selection of a choice entry based on the contents of the text, if possible. For example, with the ALPHA method, the choice entries might be selected by entering the first letter(s) of the entry. With both the NUMBER and ALPHA method, the user agent should not reorder the entries. If not specified, the default is METHOD=NUMBER.

KEY and DEFAULT

```
      o-key ::= key = identifier
  o-default ::= default = value-with-vars
```

The KEY option, if present, indicates the name of the variable to be set to the entry *value* when an entry is picked. The value of the variable comes from the choice entry (see below). An entry is picked when any action (ACCEPT, PREV, SOFT*) is selected.

The DEFAULT option indicates the default value of the variable named in the KEY option. When the card is entered, if the variable named in the KEY option is not set, it will be assigned the default value. If the variable already contains a value, the DEFAULT option is ignored.

IKEY and IDEFAULT

```
      o-ikey ::= ikey = identifier
  o-idefault ::= idefault = value-with-vars
```

The IKEY option, if present, indicates the name of the variable to be set to the entry *index* when an entry is picked. The entry index is the position of the currently selected choice entry in the choice card. An index of 0 indicates that no choice entry is selected. The IDEFAULT option indicates the default selected entry. When the card is entered, if the variable named in the IKEY option is not set, it will be assigned the default selected entry. If the variable already contains a value, the IDEFAULT option is ignored. If the IKEY option is not specified, the IDEFAULT value will be applied every time the card is entered.

On entry into a choice card, the user agent selects the initial entry in the following way.

- If the IKEY option exists, the index in the variable named by IKEY is used to select the entry. If the variable is not set, the index is assumed to be 1. If the index is larger than the number of entries in the choice card, the last entry is selected.
- If the IKEY option does not exist, and the KEY option exists, the first choice entry whose value matches the value of the variable named in the KEY option is selected. If no entry matches the value, the first entry is selected.

Once an entry is selected, the variable named by KEY is updated to the value of the entry.

Choice Entries (The CE Element)

```
      choice-entry ::= choice-entry-header choice-entry-content choice-entry-footer
 choice-entry-header ::= <CE ol(choice-entry-options) >
choice-entry-options ::= o-value | task-options
choice-entry-content ::= formatted-line
 choice-entry-footer ::= { </CE> }
```

Each choice entry displays text to the user. A choice entry is a variant of the ACCEPT action in that you can specify task-options to be used if the ACCEPT action is taken while the choice entry is selected. In addition, a choice entry can specify a value to be assigned to the variable named in the CHOICE card's KEY option.

A choice-entry-footer is optional and is implied by a subsequent <CE> or </CHOICE> tag.

An example of choice entries with task options follows:

```
 <CHOICE>
<CE TASK=GOSUB DEST="email.cgi">Email
<CE TASK=GOSUB DEST="fax.cgi">Faxes
<CE TASK=GOSUB DEST="weather.cgi">Weather Reports
</CHOICE>
```

VALUE

```
    o-value ::= value = value
```

If the VALUE option is present, it supplies the value of the variable named in the KEY option in the CHOICE element. The variable gets set when any action is selected.

The ENTRY Card Element

```
     entry-card ::= entry-header display-content entry-footer
   entry-header ::= <ENTRY ol(entry-options) >
  entry-options ::= card-options | o-key | o-default | o-format | o-noecho |
                    o-emptyok
   entry-footer ::= </ENTRY>
```

ENTRY cards let users input an optionally formatted character string. The display-content is shown the user, followed by an area to input characters. The characters entered are constrained by the entry format. An example ENTRY card for a US phone number follows:

```
<ENTRY KEY=number FORMAT="\(NNN\) NNN\-NNNN">
    Enter phone number:
</ENTRY>
```

KEY and DEFAULT

The KEY option indicates the name of the variable in the current activity to be set by this entry. It is similar to the KEY option in CHOICE cards. The entry area is pre-loaded with the value of the variable, and the value of the variable will be set to the text in the entry area when any actions (ACCEPT, PREV, SOFT*, etc.) are taken.

If the entry card is entered with the variable named in the KEY option not set, the DEFAULT option will be used to initialize the variable. If no KEY option is specified, the entry area will be initialized with the DEFAULT option, if it exists. Note, if the text entry is formatted (see below), the DEFAULT option must conform to the format. If it does not conform to the format, the behavior of the user agent is undefined.

FORMAT

```
o-format ::= format = value
```

The FORMAT option specifies the format for user input entries. The string consists of format control characters and static text which is displayed in the input area. The format control characters specify the data expected to be entered by the user. The user agent should constrain user input so that no incorrect or unexpected values can be entered. The default format is "*M".

The format codes are:

A	entry of any upper-case alphabetic character or punctuation
a	entry of any lower-case alphabetic character or punctuation
N	entry of any numeric character
X	entry of any upper-case alphabetic. numeric character, or punctuation
x	entry of any lower-case alphabetic. numeric character, or punctuation
M	entry of any MiXeD-cAsE alphabetic characters, numeric characters, or punctuation assuming first character is upper-case
m	entry of any MiXeD-cAsE alphabetic characters, numeric characters, or punctuation assuming first character is lower-case
*f	entry of any number of characters; *f* is one of the above format codes and specifies what kind of characters can be entered This format may only be specified once, and it must appear at the end of the format string.
nf	entry of *n* characters where *n* is from 2 to 9; *f* is one of the above format codes and specifies what kind of characters can be entered This format may only be specified once, and it must appear at the end of the format string
\c	display the next character, *c*, in the entry field; allows quoting of the format codes so they can be displayed in the entry area

NOECHO

```
o-noecho ::= noecho = boolean-value
```

The NOECHO option indicates whether this text entry should echo the characters entered or not. Typically, you set NOECHO=TRUE for password entry and the like. The default is NOECHO=FALSE.

EMPTYOK

```
o-emptyok ::= emptyok = boolean-value
```

The EMPTYOK option indicates whether this text entry will accept empty input even though a non-empty format string has been specified. Typically, you set EMPTYOK=TRUE for formatted entry fields that are optional. The default is EMPYTOK=FALSE.

Formatted Text

```
formatted-line ::= { { line-format } {alignment-format} text-line }
formatted-text ::= formatted-line*
     text-line ::= {text | variable | image | text-format | tab-format}*
```

Formatted text is what is shown to the user in most cards. Formatted lines are used for choice entries.

Line Formatting

```
line-format ::= <WRAP> | <LINE>
            ::= <BR>
```

The BR element is used to separate multiple lines of text. The treatment of a line too long to fit on the screen is specified by the WRAP and LINE elements. If WRAP is specified, the line is word wrapped onto multiple lines. If LINE is specified, the line is not wrapped. The user agent must provide a mechanism to view entire non-wrapped lines. An example of this is horizontal scrolling. The WRAP and LINE elements imply a line break.

When the optional line-format element is not specified, a default is assumed. The default line-format is dependent on the context of the text element. Text in the display-content portion of DISPLAY, CHOICE and ENTRY cards will default to a WRAP mode. Text in the formatted line of a choice entry (CE) element will default to LINE mode.

Alignment Formatting

```
alignment-format ::= <CENTER> | <RIGHT>
```

The alignment codes specify how the line is to be laid out. The line is either centered, left-justified, or right-justified. If not specified, all lines default to left-justified.

Text Formatting

```
text-format ::= <B> | <I>
            ::= </B> | </I>
```

The <B> and <I> elements turn on bold and italic fonts, respectively. The </B> and </I> elements cancel the formatting. These need not be strictly nested. Redundant application and cancellation should be tolerated. Formatted-text and formatted-line elements start in plain mode (no bold or italic).

Tab Formatting

```
tab-format ::= <TAB>
```

The TAB element is used to create aligned columns. Rather than tab to specific character positions, the TAB element separates the text for each column. To ensure the narrowest display width, the user agent should determine the width of the column from the maximum width of the text (or images) in each line for each column. The extent of the columns is from the first line with TAB elements through the last contiguous line with TAB elements. Some lines may have fewer TAB elements than others, in which case they are assumed to have no text for the extra columns.

Images

```
        image ::= <IMG ol(image-options) >
image-options ::= o-alt | o-name | o-src
```

Images are treated as large words and are simply displayed as part of the text. The image data is kept in a separate data stream

identified by URL or name.

SRC

```
o-src ::= src = destination
```

The SRC option specifies the URL for image. If the user agent supports images, it will download the image from the specified URL, and render it when the text is being displayed.

NAME

```
o-name ::= name = value-with-vars
```

The NAME option specifies an alternative, internal graphic representation for the image. This representation will be used if it exists, otherwise the image will be downloaded from the URL specified in the SRC option. This allows user agents to provide an internal set of generic images, (e.g., smile, sun, cloud, etc.).

ALT

```
o-alt ::= alt = value-with-vars
```

The ALT option specifies an alternative textual representation for the image. This representation will be used when the image can not be displayed, i.e. when the user agent does not support images, or the image data is not available.

The ACTION Element

```
        action ::= <ACTION ol(action-options) >
action-options ::= o-type | o-label | o-image | task-options
```

The ACTION element binds a label, an optional image, and a task to the user agents navigational user interface. The actions are specified at an abstract level, and are identified as SOFT1 through SOFT8, ACCEPT, PREV, and HELP. When the action selected, the indicated task takes place.

Actions can be specified both for the deck as a whole and per card. When specified for the deck they remain in effect for the entire deck. When specified for a card, they temporarily override any deck actions while the card is visible. Note that the override is done independently for each action type (a card can override one action while letting the deck specify the rest).

The user agent will make a best effort to provide a user interface for SOFT1-SOFT8. If necessary, SOFTx actions can be ignored by the user agent if it is not possible to provide a user interface for them.

TYPE

```
o-type ::= type = key
   key ::= soft1 | soft2 | soft3 | soft4 | soft5 |
           soft6 | soft7 |soft8 | accept | prev | help
```

The TYPE option specifies which navigational user-interface action the element applies to. The TYPE option is required.

LABEL

```
o-label ::= label = value-with-vars
```

If the user agent is able to dynamically label the user interface actions (for example if it uses soft keys), the LABEL option is the text to display for that action. The user agent may truncate or ignore the label depending on the characteristics of its navigational user interface. To work well on a broad range of user agents, is suggested that labels be 6 characters long or shorter.

IMAGE

```
o-image ::= image = url
```

The IMAGE option is the URL of the image to display for the action, if supported by the user-agent.

Destinations

Destinations are processed by the user agent in two phases. First, it parses the destination as a simple option value with variables:

```
destination ::= value-with-vars
        url ::= value-with-vars
       path ::= value-with-vars
```

Variable substitution is done in the first phase of processing.

Second, the user agent parses the destination as a URL.

```
destination ::= url { # fragment }
```

Destinations are used with GO and GOSUB tasks to indicate the next deck or card to download and display. Destinations conform to the URL specification defined in [RFC1808] and [RFC1738].

The fragment portion of the URL identifies which card in the deck to display. The name of the card (as specified in the NAME option) is used.

Tasks

Actions and choice entries can specify tasks to be performed when the user selects them.

```
task-options ::= o-task | o-dest | o-vars | o-receive | o-retvals | o-next |
                 o-cancel | o-sendreferer | o-friend | o-clear | o-confirm |
                 o-data | o-number | o-go | o-call
```

TASK

```
   o-task ::= task = task-type
task-type ::= go | gosub | prev | return |
              cancel | post | call | noop
```

The TASK option specifies which task to perform. The other options provide the information needed to perform the task. The following table describes the tasks and lists the options required for each: Options in bold are required for that task.

Task	Description	Options (required if underlined)
GO	Display the specified HDML card.	dest, vars, sendreferer
GOSUB	Push a new activity on the activity stack, and display the specified HDML card. When the sub-activity returns, put the return values into the "receive" variables and go to the "next" URL. If the sub-activity cancels, go to the "cancel" URL. Setting FRIEND=TRUE indicates that the sub-activity is friendly and is allowed to override the NEXT and CANCEL options and to unset all of the current activity's variables.	dest, vars, sendreferer, friend, receive, next, cancel
PREV	Display the previous card in the activity history. If currently displaying the first card in the activity, perform a CANCEL.	
RETURN	Return to the calling activity with the specified return values. Go to the URL in the previous activity's NEXT option. If it is not specified, go to the calling card in the previous activity. If the previous activity flagged the current activity with FRIEND=TRUE, DEST may be used to override which card to display next. Setting CLEAR=TRUE unsets all the previous activity's variables.	retvals, dest, clear
CANCEL	Cancel the current activity. Go to the URL in the previous activity's CANCEL option. If it is not specified, go to the calling card in the previous activity. If the previous activity flagged the current activity with FRIEND=TRUE, DEST may be used to override which card to display next. Setting CLEAR=TRUE unsets all the previous activity's variables.	dest, clear

CALL	Place a phone call to the specified phone number if the user agent is running on a voice phone.	number
NOOP	Do nothing. This is primarily used to disable the default behavior of actions	

Some examples:

```
<ACTION TYPE=ACCEPT LABEL="Next" TASK=GO DEST="#card2">

<ACTION TYPE=ACCEPT LABEL="Search" TASK=GOSUB
 DEST="search.cgi" RECEIVE="name"
 NEXT="email.cgi?NAME=$name">

<ACTION TYPE=ACCEPT TASK=RETURN RETVALS="John Doe">

<ACTION TYPE=PREV DEST="#cancelShield">

<ACTION TYPE=SOFT1 LABEL="Cancel" TASK=CANCEL>

<CE TASK=CALL NUMBER="411">Directory Assistance

<ACTION TYPE=SOFT1 TASK=NOOP>
```

DEST

```
o-dest ::= dest = destination
```

The DEST option specifies the URL of the card to display or invoke in the GO, GOSUB, and sometimes RETURN and CANCEL tasks.

VARS

```
o-vars ::= vars = value-with-vars
```

The VARS option specifies the variables to set in the current (in the case of GO) or sub (in the case of GOSUB) activity. The variables are specified in URL query-string format. For example:

```
vars="var1=value1&var2=value2"
```

The values should be escaped according to URL escaping conventions. The user agent will unescape the VARS option before setting the value of the variables.

RECEIVE

```
   o-receive ::= receive = receive-list
receive-list ::= { ; }* identifier { ; { identifier } }*
```

When invoking a card with the GOSUB task, the RECEIVE option specifies the names of the variables to assign the return values to. Return values are positional. For example, if you want to put the third return value into "myvar", you can specify the following RECEIVE option:

```
receive=;;myvar
```

If the sub-activity returns fewer values than assigned in the RECEIVE option, the remaining named variables without values are cleared. Return values not assigned to variables are ignored.

RETVALS

```
   o-retvals ::= retvals = retval-list
retval-list ::= value-with-vars { ; value-with-vars }
```

When returning from a sub-activity with the RETURN task, the RETVALS option specifies the values to return to the invoking activity. Return values are positional.

NEXT

```
o-next ::= next = destination
```

When invoking a sub-activity with the GOSUB task, the NEXT option specifies the destination to go to after the sub-activity returns.

CANCEL

```
o-cancel ::= cancel = destination
```

When invoking a sub-activity with the GOSUB task, the CANCEL option specifies the destination to go to if the sub-activity is canceled.

SENDREFERER

```
o-sendreferer ::= sendreferer = boolean-value
```

The SENDREFERER option specifies whether the user agent should indicate the URL of the current deck (i.e. the referring deck) when requesting the DEST, NEXT, and CANCEL decks from a server. This allows a server to perform access control on URLs based on which decks are linking to them. The URL will be the smallest relative URL possible if it can be relative at all. For example, if SENDREFERER=TRUE, an HTTP based user agent will indicated the URL of the current deck in the HTTP "Referer" request header [RFC2068]. The default is SENDREFERER=FALSE.

FRIEND

```
o-friend ::= friend = boolean-value
```

The FRIEND option is used in the GOSUB task to indicate that the sub-activity is friendly. A friendly sub-activity is allowed to use the DEST and CLEAR options in RETURN and CANCEL tasks. The default is FRIEND=FALSE.

CLEAR

```
o-clear ::= clear = boolean-value
```

The CLEAR option is used in the RETURN and CANCEL tasks to unset all the calling activity's variables. The unsetting is done before the retvals are assigned to the variables listed in the calling action's RECEIVE option. This is only available to sub-activities that have been marked as friendly by the calling activity. If the sub-activity has not been marked as a friendly, this option is ignored. The default is CLEAR=FALSE.

NUMBER

```
o-number ::= number = value-with-vars
```

The NUMBER option specifies the phone number for a CALL task.

Variables

Setting Variables

Variables are set as a side-effect of card navigation. The GO and GOSUB tasks can set variables using the VARS option. The GOSUB and RETURN tasks can set variables with the RECEIVE and RETVAL options.

When a variable is set, if the variable is already present, its value is replaced.

Variable Substitution

To substitute a variable into the deck, the following syntaxes are used:

```
  variable ::= $ identifier
           ::= $( identifier { : conversion } )
conversion ::= n { oesc }
           ::= e { scape }
```

```
variable-text ::= variable | text
```

The value of variables can be converted into a different form as they are substituted. A conversion can be specified in the variable reference following the semi-colon. The following table summarized the current conversions:

Conversion	Effect
noescape	Make no change to the value of the variable.
escape	URL escape the value of the variable.

If no conversion is specified, the variable is substituted using the conversion format appropriate for the context. The ACCESSDOMAIN, ACCESSPATH, BOOKMARK, CANCEL, DEST, GO, IMAGE, NEXT, SRC, and VARS options default to `escape` conversion, otherwise no conversion is done. Context sensitive escape conversion can be disabled with specifying the `noesc` conversion after the variable identifier.

In order to include the '$' character in text, it must be explicitly escaped, for example:

```
&dol;
```

In order to include the '$' character in options that are supposed to be URL escaped (for example, VARS), it must be explicitly escaped using URL escape form, that is:

```
%24
```

IPR2025-00857
Apple EX1014 Page 21

http://www.w3.org/TR/hdml20-7.html Go
66 captures
6 Jun 1997 - 1 Oct 2023
MAY JUN AUG
1996 06 1997 2000
About this capture

Chapter 4 : User Agent Semantics

Navigation and the Activity Stack
Deck Time-to-Live
Low-Memory Behavior
Limited History
Limited Activity Stack
Limited Cache

Some aspects of user agent behavior may not be apparent from the language specification. Such behavior is discussed herein.

Navigation and the Activity Stack

To invoke a sub-activity, an action uses the GOSUB task. This operates just like the normal GO task, only the user agent remembers the calling action on the activity stack. Specifically, the user agent needs to remember:

-
  - The URL of the calling card
  - The RECEIVE option
  - The NEXT option
  - The CANCEL option

Any card within the sub-activity can either return or cancel directly back to the calling activity. When the sub-activity returns via the RETURN task, the variables specified in the RECEIVE option are set in the calling activity, and the URL in the calling action's NEXT option is fetched and displayed. If the sub-activity returns via the CANCEL task, the URL in the calling action's CANCEL option is fetched and displayed. If the NEXT or CANCEL option is not specified in the calling action, the calling card is displayed.

A sub-activity can be marked FRIEND=TRUE by the calling action. If so marked, the sub-activity can effect the state of the calling activity. A friendly sub-activity can override the NEXT and CANCEL options by specifying a DEST option in the RETURN or CANCEL task. A friendly sub-activity can also unset all the variables in the calling activity by setting the CLEAR option to TRUE in a RETURN or CANCEL task. If both the CLEAR and RETVALS options are specified, the calling activity's variables are first cleared, and then the variables in its RECEIVE option are set from the RETVALS.

Deck Time-to-Live

If the user agent caches decks, it may continue to display cards in decks with expired TTLs under certain circumstances:

-
  - When navigating backwards
  - When navigating to a card in the same deck

The user agent is in the process of navigating backwards when it performs a PREV, RETURN, or CANCEL task. Note, however, in the case of RETURN and CANCEL if the caller has specified a NEXT or CANCEL option (respectively), the access of the NEXT or CANCEL destination is considered to be navigating forwards, i.e. normal cache expiration checking applies.

Low-Memory Behavior

Limited History

It is acceptable for the user agent to limit the number of history entries per activity. In the case of history entry exhaustion, the user agent should delete the least-recently-used history entry.

Limited Activity Stack

It is acceptable for the user agent to limit the size of the activity stack. In the case of activity stack exhaustion, the user agent should delete the oldest activity.

Limited Cache

In selecting decks to free from the cache, the user agent should refrain from freeing decks that are referenced by the activity stack. If cache space remains exhausted after freeing unreferenced entries, the user agent should delete the oldest activity in the activity stack and free any unreferenced entries in the deck cache until there is sufficient space to continue processing. However, the user agent should never delete the current entry. If cache space remains exhausted and there is only one activity on the activity stack, the least-recently-used history entry may be deleted and unreferenced cache entries freed until there is sufficient space to continue processing.

IPR2025-00857
Apple EX1014 Page 23

http://www.w3.org/TR/hdml20-8.html Go
80 captures
6 Jun 1997 - 26 May 2024
MAY JUN AUG
1996 06 1997 2000
About this capture

Appendix A : HDML 2.0 BNF Summary

```
                 deck ::= deck-header { action }* { card }+ deck-footer
          deck-header ::= <HDML o-version ol(deck-options) >
         deck-options ::= o-ttl | o-deckseq | o-public | o-accessdomain | o-accesspath | o-markable
          deck-footer ::= </HDML>

            o-version ::= version = value
                o-ttl ::= ttl = {numeric}+
             o-public ::= public = boolean-value
       o-accessdomain ::= accessdomain = url
         o-accesspath ::= accesspath = path
           o-markable ::= markable = boolean-value
        boolean-value ::= true | false

                 card ::= display-card | nodisplay-card | choice-card | entry-card
         card-options ::= o-name | o-title | o-bookmark | o-markable
               o-name ::= name = identifier
              o-title ::= title = value-with-vars
           o-bookmark ::= bookmark = destination
         display-card ::= display-header display-content display-footer
       display-header ::= <DISPLAY ol(display-options) >
      display-options ::= card-options
      display-content ::= { action }* formatted-text
       display-footer ::= </DISPLAY>

       nodisplay-card ::= nodisplay-header nodisplay-content nodisplay-footer
     nodisplay-header ::= <NODISPLAY ol(nodisplay-options) >
    nodisplay-options ::= card-options
    nodisplay-content ::= { action }*
     nodisplay-footer ::= </NODISPLAY>

          choice-card ::= choice-header display-content { entries } choice-footer
        choice-header ::= <CHOICE ol(choice-options) >
       choice-options ::= card-options | o-method | o-key | o-ikey | o-default | o-idefault

             o-method ::= method = method-type
           method-type ::= list | alpha
               entries ::= { choice-entry }+
         choice-footer ::= </CHOICE>
                 o-key ::= key = identifier
                o-ikey ::= ikey = identifier
             o-default ::= default = value-with-vars
            o-idefault ::= idefault = value-with-vars

         choice-entry ::= choice-entry-header choice-entry-content choice-entry-footer
  choice-entry-header ::= <CE ol(choice-entry-options) >
 choice-entry-options ::= o-value | task-options
 choice-entry-content ::= formatted-line
  choice-entry-footer ::= { </CE> }

              o-value ::= value = value

         entry-header ::= <ENTRY ol(entry-options) >
        entry-options ::= card-options | o-format | o-noecho | o-emptyok | o-key | o-default
         entry-footer ::= </ENTRY>

             o-format ::= format = value
             o-noecho ::= noecho = boolean-value
            o-emptyok ::= emptyok = boolean-value

        formatted-line ::= { { line-format } {alignment-format} text-line }
```

```
formatted-text ::= formatted-line*
     text-line ::= {text | variable | image | text-format | tab-format}*
   text-format ::= <B> | <I>
               ::= </B> | </I>
alignment-format ::= <CENTER> | <RIGHT>
    tab-format ::= <TAB>
   line-format ::= <WRAP> | <LINE>
               ::= <BR>

         image ::= <IMG ol(image-options) >
  image-options ::= o-alt | o-name | o-src
          o-alt ::= alt = value-with-vars
         o-name ::= name = value-with-vars
          o-src ::= src = destination

         action ::= <ACTION ol(action-options) >
  action-options ::= o-label | o-type | o-image | task-options
         o-type ::= type = key
            key ::= soft1 | soft2 | soft3 | soft4 | soft5 |
                    soft6 | soft7 | soft8 |
                    accept | prev | help
        o-label ::= label = value-with-vars
        o-image ::= image = url
```

(phase 1)

```
   destination ::= value-with-vars
           url ::= value-with-vars
          path ::= value-with-vars
```

(phase 2)

```
   destination ::= url { # fragment }

           url ::= absolute-url | relative-url
  absolute-url ::= generic-rl
               ::= scheme : { uchar | reserved }*
   generic-url ::= scheme : relative-url
  relative-url ::= net-path | abs-path | rel-path

      net-path ::= // net-loc { abs-path }
      abs-path ::= / rel-path
      rel-path ::= { path } { ; params } { ? query }

          path ::= fsegment { / segment }
      fsegment ::= { pchar }+
       segment ::= { pchar }*

        params ::= param { ; param }*
         param ::= { pchar | / }*

        scheme ::= { alpha-numeric | + | - | . }+
       net-loc ::= { pchar | ; | ? }*
         query ::= { uchar | reserved }*
      fragment ::= { uchar | reserved }*

         pchar ::= uchar | : | @ | & | =
         uchar ::= unreserved | escape
   unreserved ::= alpha-numeric | safe | extra

        escape ::= % hex hex

          safe ::= $ | - | _ | . | +
         extra ::= ! | * | ' | ( | ) | ,
      national ::= { | } | | | \ | ^ | ~ | [ | ] | `
      reserved ::= ; | / | ? | : | @ | & | =
   punctuation ::= < | > | # | % | "

  task-options ::=  o-task | o-dest | o-vars | o-receive | o-retvals | o-next |
                    o-cancel | o-sendreferer | o-friend | o-clear | o-confirm |
                    o-data | o-number | o-go | o-call
```

```
        o-task ::= task = task-type
     task-type ::= go | gosub | prev | return | cancel |
                   post | call | noop
        o-dest ::= dest = destination
        o-vars ::= vars = value-with-vars
     o-receive ::= receive = receive-list
  receive-list ::= { ; }* identifier { ; { identifier } }*
     o-retvals ::= retvals = retval-list
   retval-list ::= value-with-vars { ; value-with-vars }
        o-next ::= next = destination
      o-cancel ::= cancel = destination
o-sendreferer ::= sendreferer = boolean-value
      o-friend ::= friend = boolean-value
       o-clear ::= clear = boolean-value
        o-data ::= data = data-list
     data-list ::= identifier { ; identifier }
      o-number ::= number = value-with-vars
          o-go ::= go = destination
        o-call ::= call = value-with-vars

      variable ::= $ identifier
               ::= $( identifier { : conversion } )
    conversion ::= n { oesc }
               ::= e { scape }
 variable-text ::= variable | text

         alpha ::= any alphabetic character a through z or A through Z
       numeric ::= any digit 0 through 9
 alpha-numeric ::= alpha | numeric
           hex ::= numeric | any letter A through F, either case
   blank-space ::= { space | tab | new-line }+
         space ::= the space character
           tab ::= the tab character
      new-line ::= the carriage return character
               ::= the line feed character
               ::= the sequence carriage return, line feed

          word ::= { alpha-numeric }+
    identifier ::= alpha { alpha-numeric }*
       integer ::= { + | - } { numeric }+

          text ::= any ISO-Latin-1 character except <, >, ", &, or $
               ::= &gt; | &lt; | &quot; | &amp; |   | &dol;
               ::= &# hex hex ;

    plain-text ::= text except blank-space
```

IPR2025-00857
Apple EX1014 Page 26

http://www.w3.org/TR/hdml20-9.html Go
61 captures
6 Jun 1997 - 5 Jul 2024
MAY JUN AUG
1996 06 1997 2000
About this capture

Appendix B : Bibliography

[RFC1738]

Berners-Lee, T., et. al., "Uniform Resource Locators (URL)", RFC 1738, CERN, December 1994, http://www.internic.net/rfc/rfc1738.txt

[RFC1808]

Fielding, R., "Relative Uniform Resource Locators", RFC 1808, UC Irvine, June 1995, http://www.internic.net/rfc/rfc1808.txt

[RFC2068]

Fielding, R., et. al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, UC Irvine, January 1997, http://www.internic.net/rfc/rfc2068.txt

IPR2025-00857
Apple EX1014 Page 27