stock query over Internet **140** to a stock quote service provider using the ticker tape symbol passed as input data by server **749** to the common gateway interface application. When the response to the stock query is received, the common gateway interface application builds a PIDL deck that includes the data in the response to the stock query.

Upon completion of servicing the request, HTTP server **749** converts the PIDL deck to a TIL deck and returns the TIL deck to client module **702** using UDP in transfer response process **863**, that is connected by a dotted line to response received check **806** in client module **702**. As the TIL deck is transferred, client module **702** stores the deck in memory **716**.

After the TIL deck is transferred, HTTP server **749** closes the process for responding to the message from cellular telephone **700**. All the information needed by client module **702** to generate a user interface on display screen **705** and for responding to any selection or data entry presented in the user interface is included in the TIL deck. Consequently, client module **702** only has to interpret the TIL deck and interpret the user input to transmit the next message to HTTP server **749**. The state for the HTTP server is defined in the next message. Consequently, HTTP server **749** is stateless because HTTP server **749** does not retain state information concerning a response to a message after the message is transmitted.

However, in another embodiment (not shown), a server could retain state information concerning each interaction with a client module. For example, if the server transmitted a choice card to the client module, the server would retain state information indicating that a choice was pending from the client module. In this embodiment, when the user makes a choice, e.g., depresses key two to indicate choice two, the choice is transmitted to the server which in turn accesses the URL associated with choice two. If this URL addresses another application, the server executes that application. Thus, in this embodiment, the server retains state information concerning each interaction with a client module. In view of this disclosure, those skilled in the art can implement the principles of this invention utilizing a server that retains state information when such a client/server combination is advantageous.

Returning to the present embodiment, when the TIL deck is received, client module **702** leaves response received check process **806** and transfers to process first card **808**. However, if TCP is used instead of UDP, client module **702** upon leaving check **806** would close the virtual TCP connection in transmission completed process **807**. Upon closing the virtual TCP connection, processing would transfer to process first card **808**. Again, transmission complete process **807** is enclosed within a dashed line box to indicate that process **807** is used only with TCP.

In process first card **808**, client module **702** parses the TIL deck and interprets the first card. Processing transfers from process first card **808** to generate display process **809**.

In generate display process **809**, client module **702** passes the data to be displayed in the first card to display module **712**. Display module **712**, in response to the data, drives the text and images in the data on display screen **705**. Generate display process **809** transfers processing to key press check **820** through node **813**. In FIGS. **8**A to **8**D, any circular node with the same alphanumeric character and reference numeral is the same node. The circular nodes are used to establish connections between the various processes in the method of FIGS. **8**A to **8**D without cluttering the figures with a number of connection lines.

Client module **702** waits in key press check **820** for the user to press a key on keypad **715** of cellular telephone **700**. In this embodiment, cellular telephone **700** is assumed to have the capability to support two soft keys, a scroll-up key, a scroll-down key, a previous key, a next key, and keys zero to **9** that are configured in the standard telephone keypad configuration. In view of the following disclosure, if one or more of these keys are not present, one of skill in the art can alter the method for the particular configuration of the cellular telephone keypad, or other two-way data communication device keypad. For example, if the cellular telephone included a home key, the key press processing described more completely below would include a check that detected when the home key was pressed and would in turn transfer to get home URL process **801**.

Briefly, the processes in FIGS. **8**B to **8**C, identify the key pressed by the user, identify the action required, and then transfer to a process that implements the action required. Specifically, when a key on the keypad is pressed, keypad module **711** stores an identifier for the key in work memory **716** and notifies client module **702** of the key press. Upon receipt of the notification from keypad module **711**, client module **702** reads the storage location in work memory **716** to determine the key pressed and transfers processing from key press check **820** to scroll key check **821**.

In scroll key check **821**, client module **702** determines whether the user pressed either of the scroll keys. If a scroll key was pressed, processing transfers to adjust display process **822** and otherwise to display card check **823**.

In adjust display process **822**, client module **702** determines which of the scroll-up or scroll-down keys was pressed. Client module **702** then sends information to display module **712** so that the current display is either scrolled-up one line or scrolled-down one line. If the scroll key would move the display beyond a boundary of the current card, the scroll key press is ignored in adjust display process **822**.

In response to the information from client module **702**, display module **712** adjusts the screen display on display screen **705**. Client module **702** transfers processing from adjust display process **822** to key press check **820** through node **813**.

If a scroll key was not pressed, processing is passed through scroll key check **821** to display card check **823**. Client module **702** takes action that depends on the particular type of card that is currently being displayed on display screen **705**. If the current card is a display card, client module **702** passes through display card check **823** to soft key check **828**, and otherwise transfers to choice card check **824**.

Assuming for the moment that the current card is not a display card, choice card check **824** determines whether the current card is a choice card. If the current card is a choice card, client module **702** passes through choice card check **824** to choice key check **826**, and otherwise transfers to data key check **826**.

Assuming for the moment that the current card is neither a display card nor a choice card, the current card must be an entry card, because in this embodiment only three card types are defined. Thus, client module **702** does not check for an entry card. Rather, data key check **826** determines whether a valid data key was pressed. In this embodiment, the data keys are keys zero to nine on the key pad, and the # key. In other embodiments, other combinations of keys could be defined as data keys. If the pressed key was one of the data keys, data key check **826** transfers to process data entry **827** and otherwise transfers to soft key check **828**.

In process data entry **827**, client module **702** knows whether the predictive text entry process is turned-on, because one of the parameters on the entry card specifies

whether to use the predictive text entry process, as described in Appendix I, which is incorporated herein by reference in its entirety.

If the predictive text entry process is not turned-on, client module **702** in process data entry **827** enters the pressed key value in a text entry buffer in work memory **716** at the appropriate location. Also, client module **702** sends information to display module **712** so the value of the pressed key is displayed in the appropriate location on display screen **705** by display module **712**.

If the predictive text entry process is turned-on, client module **702** uses the novel predictive text entry process in process data entry **827**, as described more completely below with respect to FIGS. **9**, **10**A to **10**T, and **11**, to determine the letter to select from the set of letters associated with the pressed key. After the predictive text entry process determines the appropriate letter, a value representing the letter is stored at the appropriate location in the text buffer in work memory **716**. Also, client module **702** sends information to display module **712** so that the letter is displayed in the appropriate location on display screen **705**. Upon completion of process data entry **827**, client module **702** transfers processing through node **813** to key press check **820**.

The previous description assumed that the current card was an entry card, but if the current card is a choice card, choice card check **824** transferred to choice key check **826**. In generate display process **804** for the choice card, each of the choices are labeled according to information on the choice card and some or all of the choices are displayed on display screen **705**. Thus, choice key check **826** determines whether the pressed key corresponds to one of the choices. If the pressed key is one of the choices, client module **702**, in one embodiment, sends information to display module **712** to indicate the selected choice. Client module **702** also transfers from choice key check **826** through node **831** to store identifier process **850** (FIG. **8**D), that is described more completely below. Conversely, if the pressed key is not one of the choices, choice key check **826** transfers to soft key check **828**.

Soft keys can be specified both for a deck as a whole and per card, i.e., a physical key on the keypad is specified as a soft key as described more completely in Appendix I. Each soft key specification includes an identifier that defines the action to be taken when the soft key is pressed.

When a soft key is specified for a deck, the soft key remains in effect for the entire deck. However, when a soft key is specified for a card, the card soft key specification temporarily overrides the corresponding deck soft key specification, i.e., the deck soft key specification for the same physical key as the card soft key specification, while the card is visible, i.e., displayed on display screen **705**. This override is done independently for the two soft keys. Thus, soft key check **828** transfers processing to first soft key check **829** if the key pressed is one of the two possible physical soft keys. Conversely, soft key check **828** transfers processing to next key check **840** (FIG. **8**C), if neither of the two possible physical soft keys is pressed by the user.

In first soft key check **829**, client module **702** determines whether the pressed key corresponds to the first soft key. If the pressed key is the first soft key, check **829** passes the active identifier for the first soft key to store identifier process **850** through node **831**. Conversely, if the pressed key is not the first soft key, processing transfers from check **829** to second soft key check **830**.

If the pressed key is the second soft key, check **830** passes the active identifier for the second soft key to store identifier process **850** through node **831**. Conversely, if the pressed key is not the second soft key, e.g., a physical key that can be defined as a soft key was pressed but neither the current deck nor the current card defines a soft key for that physical key, processing transfers from check **830** to key press check **820** through node **813**.

When pressing transfers to next key check **840**, client module **702** determines whether the pressed key was the next key. If the next key was pressed, processing transfers to display card check **841** and otherwise to previous key check **846**.

If a display card is the current card, the next key is used to move to another card in a deck, or alternatively to another deck. Thus, display card check **841** transfers processing to last card check **842** when a display card is the current card, and otherwise to entry card check **843**.

Last card check **842** determines whether the current card is the last card in the deck. If the current display card is not the last card in the deck, last card check **842** transfers processing to read next card process **845**, which in turn reads the next card in the deck and transfers through node **812** to generate display process **809**.

If the current display card is the last card in the deck, the deck includes an identifier that specifies the location to transfer to from the last card. This identifier can be a URL to another deck, to a common gateway interface program, or an address for a card within the current deck, for example. Thus, last card check **842** transfers through node **831** to store identifier process **850** when the current display card is the last card in the deck.

If the current card is not a display card but is an entry card, display card check **841** transfers to entry card check **843**. In this embodiment, the next key is the predetermined key used to indicate that all the data for an entry on an entry card has been entered. Thus, if the current card is an entry card, entry card check **843** transfers processing to store data process **844**.

Store data process **844** stores the data entered in at an appropriate location in memory that is specified in the current entry card. Typically, the data is combined as an argument with a URL and stored. Upon completion, store data process **844** transfers through node **810** to create HTTP request process **802** (FIG. **8**A).

When the next key is pressed, if the current card is neither a display card nor an entry card, the current card is a choice card. However, as indicated above, in this embodiment client module **702** requires that the user make a choice and does not allow use of the next key. Consequently, if the current card is not an entry card, entry card check **843** transfers processing through node **813** to key press check **820**.

The previous discussion assumed that the next key was pressed and so next key check **840** transferred processing to display card check **841**. However, if the next key was not pressed, next key check **840** transfers processing to previous key check **846**. If the previous key was pressed, check **846** transfers to first card check **847** and otherwise returns processing to key press check **820**.

First card check **847** determines whether the current card is the first card of a deck. If the current card is not the first card, processing transfers from first card check **847** to read previous card **849**, which in turn reads the previous card and transfers to generate display process **809** through node **813**. Conversely, if the current card is the first card, processing transfers to home deck check **848**.

If the current card is the first card in the home deck, there is not a previous card and so home deck check transfers processing to key press check **820** through node **813** and so

the previous key press is ignored. If the current deck is not the home deck, home deck check **848** retrieves the identifier for the previous deck and transfers through node **831** to store identifier process **850**.

Store identifier process **850** is reached through node **831** from several different points. The operations in store identifier process **850** are the same irrespective of the particular process that transfers to process **850**. In each instance, an identifier is passed to store identifier process **850** and process **850** saves the identifier in working memory **716**. The identifier can be, for example, a pointer to another location in the current card, an address of another card in the current deck, a URL to a deck stored in working memory **716**, a URL to a TIL deck in TIL decks **760** on computer **743**, or perhaps, a URL to a common gateway interface program in CGI programs **761** on computer **743**. Thus, process **800** checks the stored identifier to determine the action required.

Specifically, in identifier to current deck check **851**, client module **702** determines whether the identifier is to a card in the current deck. If the identifier points to the current deck, check **851** transfers processing to retrieve data process **852** and otherwise to URL to local deck check **853**.

In retrieve data process **852**, client module **702** retrieves the information stored at the location indicated by the identifier from working memory **716** and processes the information. Retrieve data process **852** transfers through node **812** to generate display **809** (FIG. **8**A) that was described above.

URL to local deck check **853** determines whether the identifier is a URL to a deck that is stored in working memory **716**, e.g., cached. If the deck is stored locally, check **853** transfers to retrieve local deck **854** which in turn moves the local deck into the storage location for the current deck. Retrieve local deck **854** transfers processing through node **811** to process first card **808** (FIG. **8**A), that was described above.

If the identifier is neither to a location in the current deck, nor to a local deck, the identifier is a URL to an object on computer **743**. Thus, in this case, check **853** returns processing to create HTTP request **802** through node **810**.

Process **800** continues so long as the user continues to enter and process the information provided. In this embodiment, process **800** is terminated, for example, either by the user powering-off cellular telephone **700**, selecting a choice or entry card that discontinues operations of client module **702**, or remaining inactive for a time longer than a time-out period so that client module **702** shuts itself down.

To further illustrate the operations in process **800**, consider the following example which is returned to client module **702** as a TIL deck in response to a HTTP request generated by process **802**. For readability, Table 2 presents the deck in PIDL. In this example, all of the choices are for applications on the same server. However, in another embodiment, each URL could address any desired combination of servers.

TABLE 2

EXAMPLE OF PIDL CHOICE DECK

```
<PIDL>
<CHOICE>
<CE URL=http://www.libris.com/airnet/nnn>News
<CE URL=http://www.libris.com/airnet/www>Weather
<CE URL=http://www.libris.com/airnet/sss>Sports
</CHOICE>
</PIDL>
```

In process first card **808**, client module **702** interprets the information in Table 2 and transfers to generate display process **809**. In generate display process **809**, client module **702** sends information to display module **712** so that the user is presented with a list of three choices on display screen **705**, i.e, a user interface for the choice card is generated:

1. News
2. Weather
3. Sports

Generate display process **809** (FIG. **8**A) transfers to key press check **820** (FIG. **8**B). When the user presses the two key on keypad **715**, key press check **820** transfers through check **821** to display card check **823**.

Since the current card is a choice card, check **823** transfers processing to choice card check **824**, which in turn transfers to choice key check **826**. Since the two key was pressed and that key is a choice key, check **826** transfers processing to store identifier process **850** (FIG. **8**D). In process **850**, client module **702** stores the URL corresponding to two, i.e,

URL=http://www.libris.com/airnet/www

in working memory **716**.

Since this URL is to an object on computer **743**, processing transfers through checks **851** and **853** to create HTTP request process **802**, which in turn generates the request. When the HTTP request is transmitted to server **749**, as described above with respect to process **804**, server **749** in service request process **862** retrieves deck www from TIL decks **760**. An example of the deck is given in Table 3. Again for readability, the deck in present herein in PIDL.

TABLE 3

EXAMPLE OF A SECOND PIDL CHOICE DECK

```
<PIDL>
<CHOICE>
<CE URL=http://www.libris.com/airnet/www-1>World
<CE URL=http://www.libris.com/airnet
                /www-2>National
<CE URL=http://www.libris.com/airnet/www-3>State
<CE URL=http://www.libris.com/airnet/www-4>Local
</CHOICE>
</PIDL>
```

The deck in Table 3 is transmitted to cellular telephone **700** and stored in memory **716**, as described above with respect to process **806**. The choice card is processed in process **808** and displayed in process **809**. As a result of process **809**, the user is presented with a list of choices:

1. World
2. National
3. State
4. Local.

When the user makes another selection, the same sequence of processes as described above for the first choice card is executed by client module **702**, and another URL is stored that points to a program on server **749** that retrieves the desired weather information and generates a deck with that information. This deck is transferred to cellular telephone **700** and displayed.

As described above, if the current card is an entry card and a key is pressed, client process **702** reaches data key press check **826** (FIG. **8**B). If the pressed key is a valid data key, check **826** transfers to process data entry **827**.

In one embodiment, process data entry **827** uses a novel predictive text entry process for text entry. Recall that on a typical telephone keypad, the keys are labeled with both a number and two or three letters. For example, the two key

is also labeled abc. This leads to some ambiguity when using the telephone keypad to enter text. Is the user attempting to enter an a, b, or c when the two key is pressed?

In one prior art method, two keystrokes were required to enter each letter of text. The first keystroke identified the first key and the second key stroke identified the specific letter desired on the first key. For example, to enter the letter s, the user would first press the seven key that is labeled with letters p, r, and s. Next, the user would press the three key to select the letter s. While this method may work well for short sequences that consist of only three or four letters, the method does not work well for English text. For example, if the user has already entered th and then presses the three key that is labeled with letters d, e, and f, almost always the desired next letter is the letter e. Therefore, making the user press the two key is an extra and unnecessary step.

Client module **702** of this invention utilizes a novel predictive text entry process to reduce the number of key strokes required to enter text using a telephone keypad, or any similar keypad. Using this process, in most cases a single key stroke suffices to enter a single letter.

While this embodiment of the invention is described in terms of a telephone keypad, the principles of the invention are not limited to only a telephone keypad. In general, the process described more completely below, can be extended to any keypad where a single key is used to enter two or more letters. Further, the process is not limited to only letters, but rather is applicable to any keypad where a single key is used to represent two or more characters. In view of the following disclosure, those skilled in the art can use the principles of the predictive text entry process in a wide variety of applications.

The system for predictive text entry includes a predictive text entry module **901** that in this embodiment is included in client module **702**, keyboard module **711**, and a letter frequency table **902** that is loaded into memory **716**, when client module **702** is activated. Predictive text entry module **901** is used in process data entry **827** when specified by the current entry card. Predictive text entry module **901** performs routine buffer management processes, that are known to one of skill in the art and so are not described further to avoid detracting from the process.

Predictive text entry module **901** stores a letter entry for each letter entered in a text buffer **903** in memory **716**. In this embodiment, letters Q and Z are assigned to the one key and the zero key is used to enter a space, period, and comma, i.e., the zero key provides punctuation. However, these assignments are illustrative only, and are not intended to limit the invention to this particular embodiment.

The first letter entered is placed at the left end of the buffer and each additional letter is placed in the left most unused space in buffer **903**. Thus, the last letter entered in text buffer **903** is the right most character. Letter frequency table **902**, sometimes referred to as a table of predictive letter entries, is a look-up table where each entry in the look-table is addressed by three indices. The first two indices represent the two most recently entered letters in text buffer **903** and the third index represents the key that was pressed. Each predictive letter entry stored in letter frequency table **902** defines which of the letters associated with the pressed key to use given the previous two letters. For example, since the is a commonly occurring string, the entry in table **902** addressed by (t, h, 3) returns e, or more concisely the predictive letter entry **2** is returned to indicate that the second letter of the group of letters d, e, and f associated with the three key is the predicted letter. Of course, letter frequency table **902** could be altered to return more than a single letter.

In this embodiment, letter frequency table **902** was empirically generated using a collection of e-mail. Appendix II is a computer program listing that was used to generate letter frequency table **902** that is illustrated in FIGS. **10**A to **10**T. Briefly, the computer program implements a process that sequentially steps through the data provided and (i) for each possible single letter determines the most likely letter that follows for each key on the keypad; and (ii) for each possible combination of two letters determines the most likely letter that follows for each key on the keypad. In this embodiment, the most likely letter is the letter having the greatest frequency after the single letter. Similarly, the most likely letter is the letter having the greatest frequency after the combination of two letters. If there is a tie in the frequency, the first letter associated with a key is selected. Of course, other measures of likelihood could be used to generate the entries in table **902**.

Thus, in FIGS. **10**A to **10**T, the first of the ten columns, i.e., the left most column, is the two letter sequence and the first row, i.e., the top row is the keys on the key pad used to enter text. A combination of an entry in the first column and a key in the top row is used to select the predicted text entry. Thus, using the example of th, this two key sequence appears in the first column of FIG. **10**O. When the three key is pressed, the letter in the row with th as the first entry and in the column with three as the first entry, i.e., e, is retrieved. Alternatively, if the four key is pressed, letter i is retrieved from the table.

In this embodiment, table **902** is a buffer of two bit numbers. Each two bit number has a value in the range of zero to three, and the two bit number represents a predicted letter for the pressed key. Thus, for a two key labeled with letters A, B and C, a zero represents A; a one represents B; and a two represents C. In general, the number of bits used is determined by the key that represents the maximum number of characters. In this embodiment, the maximum number of characters represented by a key is three. The number of storage bits required is an integer S where S is the smallest number such that **2****S is greater than or equal to the maximum number of characters represented by a key.

In this embodiment, three indices i0, i1, and i2 are used generate a table index that in turn is used to access a particular predictive letter entry in table **902** of two bit numbers. Each letter is represented as a number, i.e., a letter entry, with letter A being zero, letter B being a one, letter C being a two, and so forth with letter Z being twenty-five. A space element is assigned a space element value of twenty-six. Thus, in this embodiment, there are twenty-seven possible characters.

Upon the initial entry to process **1100** (FIG. **11**), letter indices i**0**, i1, and i2 were set to twenty-six in the initial processing of the entry card to indicate that the text buffer is empty. Also, as explained more completely below, as each letter of text is entered, letter indices i0 and i1 are updated and stored in memory **716**.

However, in another embodiment, an initialize indices process is the first operation in predictive text entry process **1100**. In this embodiment, for the first letter entered, letter indices i0 and i1 are set to twenty six; for the second letter entered, letter index i0 is set to twenty six and letter index i1 is set to the value of the letter in text buffer **903**; and for all letters entered after the first two, the value associated with next to the last letter in text buffer **903** is assigned to letter index i0 and the value associated with the last letter in text buffer **903** is assigned to letter index i1.

Punctuation key check **1101** determines whether the zero key was pressed, i.e., the key selected to represent punctuation.

If the zero key was pressed, processing transfers from check **1101** to process punctuation entry **1102**. Process punctuation entry **1102** sets index i2 to twenty-six, and sends the space element value to display letter process **1108**. Display letter process **1108** transfers the space element value to display module **712** which in turn drives a space in the text entry on display screen **705**. This completes the operation of process data entry for a zero key press and so processing returns to key press check **820**.

If the zero key was not pressed, processing transfers through punctuation key check **1101** in data entry process **1100** to key one-to-nine check **1103**, i.e., to a data entry key check. If the pressed key was any one of keys one to nine, check **1103** transfers to set letter index process **1104** and otherwise to rotate last entry process **1109**.

In set letter index process **1104**, one is subtracted from the numeric value of the pressed key and the resulting value is assigned to index i2 . Set index process **1104** transfers to generate table index process **1105**.

Generate table index process **1105** combines indices i0, i1 and i2 to create a table index. In this embodiment, table index TABLE_INDEX is defined as:

$$TABLE\_INDEX=(((i0*27)+i1)*9)+i2$$

Upon completion of generate table index process **1105**, generate text entry process **1106**, retrieves the two bit value in the table at the location pointed to by table index TABLE_INDEX and converts the two bit value to a letter represented by the two bit value.

Generate text entry process **1106** transfers to update index process **1107**, which in turn stores the value of letter index i1 as letter index i0; stores the value of the retrieved letter in letter index i1; and stores the predicted letter in text buffer **903**. While this step assumes that letter indices i0, and i1 are stored and accessed each time in process **827**, alternatively, the last two letters in text buffer **903** can be retrieved and assigned to indices i0 and i1, respectively, as described above.

Update index process **1107** transfers to display letter process **1108**. Display letter process **1108** sends information to display module **712** which in turn generates the predicted letter on display screen **705**.

If the pressed key is not one of keys one to nine, i.e, is not a data entry key, processing transfers from check **1103** to rotate last entry **1109**. Recall that data key check **826** determined whether the pressed key was one of the zero to nine keys, or the # key. Thus, since checks **1101** and **1103** determined that keys zero to nine were not pressed, the only key press remaining is the # key, i.e., the rotate entry key, which indicates the user wants a letter different than the one entered last in text buffer **903**. In rotate last entry **1109**, the last character, i.e., the right most character, in text buffer **903** is replaced by the next character in the set of characters assigned to the last key pressed before the # key was pressed. Again, the use of the # key is illustrative only and is not intended to limit the invention to the use of that particular key to rotate an entry.

For example, if the last character in the text buffer **903** was a t and the # key is pressed, process **1109** changes the t to u. If the # key is pressed again, the u is changed to a v. Alternatively, if the last character in text buffer **903** was a u and the # key is pressed, process **1109** changes the u to a V. If the last character in text buffer **903** was a v and the # key is pressed, process **1109** changes the v to a t. If index i1 is stored, as the last character in text buffer **903** is rotated, index i1 is updated.

Text entry in cellular telephone **700** in different languages or contexts can be supported by using different letter frequency tables. For example, for plumbers, the prediction table can be based on text about plumbing procedures. For Frenchmen, the prediction table can be based on French text. Also, multiple letter frequency tables could be stored in cellular telephone **700**, or selectively transmitted to cellular telephone **700**, and a particular letter frequency table would be selected on an entry card.

In addition, an entry in the table can be more that a single letter, and thus save even more key strokes. For example, if the text buffer contains sche then typing a 3 could return dule rather than just d. Further, this novel method of text entry can be utilized with other than a cellular telephone. The method is applicable to any device that has several characters assigned to a single key on a keypad.

In the above embodiment, the English alphabet and a space element were used as the character set. Thus, the number 27 used in defining the table index is just the number N of characters in the set. Similarly, the number 9 used in defining the table index is just the number M of keys in the keypad that represent two or more different characters. Hence, predictive text entry method of this invention is not limited to text and is directly applicable to any keypad where each key represents a plurality of different characters.

In the embodiment of FIGS. **7**, **8**, and **9**, client module **702** and server module **749** communicate over CDPD network **710**. However, this architecture is illustrative only of the principles of the invention and is not intended to limit the invention to the particular architecture described. Client module **702** and server module **749** can use a wide variety of two-way data communication links to exchange resource locators, e.g., URLs, and TIL decks. For example, the communications link could be a switched voice circuit in which the client module and server module communicate using modems. Alternatively, the communications link could be any other packet switched network, so long as there is some way for client module **702** to get requests to server module **749** and for server module **749** to send data back to client module **702**. Further, a special purpose server could be used in place of HTTP server **749**. For example, the principles of this invention can be used over various data transport mechanisms including circuit switched data and packet switched data. These data transport mechanisms are being defined and implemented for most of the cellular network standards including GSM, TDMA, and CDMA.

In the configuration of airnet network **750** (FIG. **7**), client module **702** communicated directly with a server computer **743**. In another embodiment, as illustrated in FIG. **5**, the two-way data communication device first communicates with an airnet network translator **500** that in turn communicates with the appropriate server. In this embodiment, the operation of two-way data communication devices **100**, **101**, and **102** is similar to that described above for cellular telephone **700**, except the method field in the request generated in process **802** has a different form. For example, using the same information as before, the method field in this embodiment is:

```
GET http://www.libris.com/airnet/home.cgi?&cost=1 ANTP/1.0
```

The method field includes the full address of the server, the expected cost of the service, and the version of the protocol used for communicating with airnet network translator **500**. The two-way data communication device transmits the HTTP request including the complete URL to airnet network translator **500**.

FIG. **12** is a more detailed block diagram that illustrates the structures in one embodiment of airnet network translator **500**, according to the principles of this invention. In

this embodiment, airnet network translator **500** is a computer running under the UNIX operating system with an interface to CDPD network **710**. Such computers are well known to those skilled in the art. Thus, herein only the structures and processes that must be added to such a computer are described.

Airnet network translator **500** supports internet protocol (IP) connections over CDPD network **710** and with each computer network with which translator **500** can interact. In this embodiment, each of the modules in network translator **500** are processes that are executed by the processor in the computer. Control module **1201** is a daemon that listens for transmissions over an IP connection from CDPD network **710**. When control module **1201** accepts a transmission, control module **1201** spawns an ANT request processor **1204**, which in this embodiment is a process, as indicated above. While in FIG. **12**, only one ANT request processor **1204** is shown, there is an ANT request processor spawned for each transmission that control module **1201** accepts and the ANT request processor remains active until the communication is terminated.

FIG. **13** is a process flow diagram that illustrates the operation of ANT request processor **1204**. This process flow diagram considers transmissions that utilize both TCP/IP and UDP/IP. However, the processes that are specific only to TCP/IP are enclosed in dashed-line boxes. Upon being spawned for a TCP/IP, in establish connection process **1300**, ANT request processor **1204** establishes a TCP connection using a TCP module in the server with the client module over CDPD network **710**. After the connection is established processing transfers from process **1300** to request received check **1301**.

If UDP is being used, upon being spawned ANT request processor **1204** initiates processing in request received check **1301**. In check **1301**, ANT request processor **1204** determines whether the request from cellular telephone **700** (FIG. **12**) has been received and stored in memory **1210**. Memory **1210** represents both RAM and non-volatile memory in this embodiment. When the request has been received and stored, processing transfers from check **1301** to retrieve data process **1302**.

In retrieve data process **1302**, ANT request processor **1204** retrieves information concerning the source of the URL, i.e., client module **702** of cellular telephone **700** from customer database **1213**, and the destination specified in the URL, i.e., the designated server, from server database **1212**. Both databases **1212** and **1213** are stored in memory **1210**. A customer record in database **1213** includes, for example, a carrier address, e.g., an IP number, an airnet network translator account number, billing information, and server subscriptions. A server record in database **1212** includes a server IP address, name, category, and class of service. Class of service refers to the pricing of the service, e.g., basic services, premium services, or pay-per-view services. Other pricing schemes can be supported in other implementations. When the information is retrieved for the server and service specified in the URL, and for the customer, processing transfers to valid request check **1303**.

In valid request check **1303**, ANT request processor **1204** determines, for example, whether client module **702**, i.e., the customer, is authorized to access airnet network translator **500**; whether client module **702** is authorized to access the server specified in the URL; whether the specified server is available through translator **500**; and whether the specified server supports the requested service. Thus, valid request check **1303**, validates the client, the server, and the client/server pair. Also, since an estimated cost is included in the request, the status and credit limits on the customer's account could be checked to determine whether the estimated cost is acceptable. If all of the checks are true, processing transfers to create HTTP request process **1306**. Conversely, if any one of the checks is untrue, valid request check **1303** passes information concerning the error to return error process **1304**.

Return error process **1304** launches a CGI program stored in memory **1210** based on the information received and passes appropriate information to the CGI program. The CGI program builds an appropriate PIDL deck describing the error and converts the PIDL deck to a TIL deck, as described above. When the TIL deck describing the error is complete, return error process **1304** transfers processing to log transaction process **1315** that is described more completely below.

If all the checks in valid request check **1303** are true, create HTTP request **1306** converts the request in memory **1211** to a request specific to the server specified, which in this embodiment is a HTTP request. For example, for the above request, create HTTP request process **1306** generates a method field, such as

```
GET/airnet/home.cgi?&client=xyz&cost=1 HTTP/1.0
```

In this embodiment, the method field includes the same information as in the embodiment described above, and in addition, the method field includes a client identification and the estimated cost.

After create HTTP request process **1306** is complete, ANT request processor **1204** accesses TCP module **1203** in establish server connection process **1307** for TCP/IP and transfers to secure transmission check **1308** for UDP/IP. In establish connection process **1307**, a connection is made between the server designated in the client request and the TCP interface module (not shown) so that data can be transmitted between airnet network translator **500** and the server. When the TCP connection to the server is established, ANT request processor **1204** transfers processing from establish server connection process **1307** to secure transmission check **1308**.

In secure transmission check **1308**, ANT request processor **1204** determines whether the HTTP request from the client requested a server that utilizes a protocol that supports encryption. If such a server was requested, processing transfers to negotiate process **1309** and otherwise to transmit request process **1310**.

In negotiate process **1309**, ANT request processor **1204** negotiates an encryption technique with the server. Upon completion of the negotiation, processing transfers from process **1309** to encryption process **1311**. In encryption process **1311**, the HTTP request is encrypted using the negotiated encryption technique, and then processing transfers to transmit request process **1310**.

In transmit request process **1310**, the HTTP request is sent from memory **1210** to the HTTP server. When the transmission is complete, ANT request processor **1204** goes to result received check **1312**.

As described above, upon receipt of the request, the HTTP server services the request. Upon completion of servicing the request, the HTTP server returns either a PIDL deck or a TIL deck to airnet network translator **500**. The deck is stored in memory **1210**. If the server does not convert the PIDL deck to a TIL deck, the translation is done by airnet network translator **500**.

When the deck is received and stored, ANT request processor **1204** transitions from check **1312** to transmission

completed process **1313** for TCP/IP and to secure transmission check **1314** for UDP/IP. ANT request processor **1204** closes the TCP circuit with the server in transmission completed process **1313**. Upon closing the server TCP connection, processing transfers to secure transmission check **1314**.

If the server utilized encryption, the deck stored in memory **1210** is encrypted. Thus, secure transmission check **1314** transfers processing to decryption process **1316** if encryption was used and otherwise to log transaction **1315**.

In decryption process **1316**, the encrypted deck is decoded and stored in memory **1210**. Also, after the decoding, if the deck must be converted to a TIL deck, the translation is performed. Decryption process **1316** transfer to log transaction process **1315**.

In log transaction process **1315**, ANT request processor **1204** writes a description of the transaction to transaction log **1211** in memory **1210**. In this embodiment, each transaction record includes a customer identification, a server identification, time required for the transaction, cost of the transaction, and a completion code. In one embodiment, for security purposes, each cellular telephone is assigned to only one customer and only one account.

After the transaction is logged, processing transfers to transmit result **1317**. In transmit result **1317**, ANT request processor **1204** returns the deck to client **702**. After the deck is transmitted, ANT request processor **1204** is terminated.

In one embodiment, if an airnet network translator is fully loaded and another transmission comes in, the translator returns the address of another airnet network translator and refuses the transmission. The cellular telephone transmits the message to the other airnet network translator. In yet another embodiment, all incoming transmissions are directed to a router. A plurality of airnet network translators are connected to the router. The router monitors the status of each translator. Each incoming transmission is routed to the least busy translator, which in turn responds to the transmission and performs the necessary operations for continuing communications with the client module.

In the above description of client module **702**, module **702** interacted with components within the cellular telephone to perform the various operations specified by the user. To insulate client module **702** from the exigencies of various cellular telephones to the extent possible, a general architecture for client module **702** is described more completely below. This general architecture is designed to have specific manager modules that interact with the modules described above within the cellular telephone and to provide standard information to the remaining manager modules within client module **702**. The manager modules with client module **702** form an interpreter that interprets TIL decks to generate a user interface; interprets data input by the user; and interprets the TIL decks so that the data input by the user is combined with an appropriate resource locator and either a message is sent to an appropriate server, or another local TIL deck is interpreted by client module **702**. While this embodiment is for a cellular telephone, the manager modules are generic and so are applicable to any client module in a two-way data communication device.

This approach limits the modifications that must be made to client module **702** to implement the principles of this invention in a wide variety of two-way data communication devices over a wide variety of two-way data communication networks. Also, in the above embodiment, client module **702** supported communications and interactions over the cellular telephone network. However, client module **702** can also support local services on cellular telephone **700**. Typical local services includes local messages, an address book, and preconfigured e-mail replies, or any combination of such services.

In this embodiment, client module **702** includes a plurality of manager modules including a navigation manager module **1401**, a network manager module **1402**, a TIL manager module **1403**, an archive manager module **1404**, a local manager module **1405**, an event manager module **1406**, a timer manager module **1407**, a user interface manager module **1408**, a memory manager module **1409**, and a device dependent module **1410**.

Navigation manager module **1401** handles card and deck navigation as well as managing any caches. Navigation manager module **1401** owns and manages a history list and as well as a pushed card list. In addition, navigation manager module **1401** functions as the main line of client module **702**; does all event distribution; and supports local services.

For local services, like local message store, there are two basic approaches that can be used. First, local services are implemented in a CGI-like manner. Each local service has an entry point which is called with an argument list. A TIL deck is returned via the event manager. From that point on, the TIL deck is processed in the standard manner. This approach limits local services to the same constraints as remote services. A less restrictive approach is to allow the local service to field events instead of the standard event loop. The local service would construct TIL cards on-the-fly and feed them to user interface manager **1406**. Note that the local service would need to cooperate with the standard event loop with regard to the history, the pushed card list, and any other state that is normally managed by the event loop. Table 4 is a listing of processes for the architecture is for navigation manager module **1401**.

TABLE 4

ARCHITECTURE FOR NAVIGATION MANAGER MODULE 1401

```
ProcessEvents (void);
PushLocation (void * location, Boolean forStack);
void * PopLocation (Boolean forStack);
void * CurrentLocation();
struct LOCAL_ SERVICE {
    char name[50];
    FUNC HandleEvent(Event * pevent);
    FUNC StartLocalService(void);
    FUNC StopLocalService(void);
    };
static LOCAL_SERVICE localServices[]={ . . . };
STATUS HandleEvent(Event * pevent);
STATUS StartLocalService();
STATUS StopLocalService();
```

Routine ProcessEvents is the main entry point for event processing in client module **702**. Typical events include key presses on the keypad, choice selection for a choice card, text entry for an entry card, network events, and history events. Routine ProcessEvents can be called at any time to process an event or events. Routine ProcessEvents does not return until all events on a queue generated by event manager module **1406** are processed. If a local service is running, events are distributed to the local service before being processed by routine ProcessEvents.

The remaining routines in Table 4 are called internally to navigation manager module **1401** and by local services. Routine PushLocation pushes a location on the history list and issues a request for that location. The forStack flag indicates a stack push of local cards.

Routine *PopLocation pops a location on the history stack and issues a request for the top location of the history stack.

In routine *PopLocation the forStack flag indicates that all cards since the last stack push should be popped.

Routine *CurrentLocation returns the current location the current URL being displayed.

As shown in Table 4, each local service provides a number of functions. If a local service is running, function HandleEvent, the local service's event handler, is called before any processing by navigation manager module **1401**. If the event is handled by the local service, the event is not processed any further.

Function StartLocalService is the local services start function. Function StartLocalService is called before any events are distributed to the local function. Similarly, function StopLocalService is the stop function for the particular local service. Function StopLocalService is called when no more events are distributed to the local service.

Network manager module **1402** insulates the rest of client module **702** from the specific networking protocol used over the cellular telephone network. Network manager module **1402** delivers requests to the server specified in the URL via the cellular telephone network interface; segments responses from the server for lower latency; delivers responses from local services to navigation module **1401** via event module **1406**; handles request/response cycle (e.g. cancellation, retry strategy) with the server over the cellular telephone network; can receive asynchronous messages from the server; performs memory management of TIL decks; performs caching of TIL decks; handles all negotiations concerning protocols and server scaling with the server; handles any encryption for information exchanged between cellular telephone **700** and the server.

In some cellular telephone, the maximum message size is fixed. However, for UDP and TCP messages, a more direct interface is used that bypasses this limitation of message passing. It is important to avoid copying network data from memory buffer to memory buffer as such copying increases the memory "high water mark" as well as decreases performance. Since different cellular telephones have different interfaces for delivering network data, network manager module **1402** manages the network data. In this way, network data is only copied from the network buffer for long-term storage.

When a message or reply arrives, network manager module **1402** uses event manager module **1406** to report that fact. However, access to the data by other manager modules in client module **702** is through a protocol that allows storage of data in a variety of fashions on different telephones. Any transparent, short-term caching of TIL data is handled by network manager module **1402**. Table 5 is one architecture for network manager module **1402**.

TABLE 5

SPECIFICATION FOR NETWORK MANAGER MODULE 1402

typedef short TID;
void NM_Init(void);
void NM_Terminate(void);
TID NM_SendRequest (void *requestData, int length, Boolean ignoreCache);
NM_CancelRequest (TID TRANSACTIONId);
NM_DataType(TID TRANSACTIONId);
NM_GetData(TID TRANSACTIONId, void *data, int *length, Boolean *complete);
void *NM_HoldData (TID TRANSACTICNId);
NM_ReleaseData(TID TRANSACTIONId);
TID NM_StartData(int data Type, char *requestData, int length);
STATUS NM_EndData(TID TRANSACTIONId);

TABLE 5-continued

SPECIFICATION FOR NETWORK MANAGER MODULE 1402
STATUS NM_SetDataLength (TID TRANSACTIONId, int length); STATUS NM_GrowDataLength (TID TRANSACTIONId, Int grow); int NM_GetDataLength(TID TRANSACTIONId); void *NM_GetDataPointer (TID TRANSACTIONId); STATUS NM_DeliverData (TID TRANSACTIONId);

Network manager module **1402** identifies each network data transaction by a 16-bit transaction identification code TID. Network manager module **1402** increments transaction identification code TID by one for each new transaction. Transaction identification code TID rolls over after 0xffff.

Routine NM_Init initializes network manager module **1402** and so is called before any other calls in network manager module **1402**. Routine NM_Terminate closes processing of network manager module **1402** and so is called after all other calls in network manager module **1402**.

Network manager module **1402** uses routine TID NM_SendRequest as the standard process of sending a request to the server. Pointer *requestData in the call to routine TID MN_SendRequest is defined by the server protocol. Similarly, the state, e.g., the Boolean value, of variable ignoreCache is used to indicate whether any cached replies should be ignored. After sending the request, this routine returns a server transaction identification code TRANSACTIONId. A local service can also send a request to the server.

When the user instructs client module **702** to cancel a request, network manager module **1402** calls a routine NM_CancelRequest with cellular telephone transaction identification code TID and server transaction identification code TRANSACTIONId. Routine NM_CancelRequest issues a command to the server to cancel the specified request.

When data are received from the network, the data can be either a response to a request sent by routine TID MN_SendRequest, or by a local service. Thus, in response to receiving data from the server, network manager module **1402** generates an event that includes server transaction identification code TRANSACTIONId and the type of data DATAType. For replies to requests sent by routine TID MN_SendRequest, server transaction identification code TRANSACTIONId is the same as the one returned by the matching call to routine TID MN_SendRequest and data type DATAType indicates that the data is a response. For local service originated messages, server transaction ID is new, and data type DATAType depends on whether the data is an e-mail, pushed TIL, or another type.

After the network event is received by event manager module **1406**, and navigation manager module **1401** distributes control of the event to network manager module **1402**, network manager module **1402** users the server transaction identification code TRANSACTIONId and the remaining routines in Table 5 to process the data.

Routine NM_DataType is used to return the particular data type dataTYPE, e.g, reply, MIME, server push, etc. Routine NM_GetData sets a pointer to the data identified by server transaction identification code TRANSACTIONId, retrieves the length of the data, and determines whether all the data has been received. The interface provided by this routine allows the first part of a data stream, e.g. the first card of a TIL deck, to be processed by client module **702** before the rest of the deck is received.

Routine NM_HoldData is called before calling routine NM_GetData to hold the data and thus insure that the data remains valid during processing by client module **702**. If the data is not held, the data can be deleted or moved with the internal buffers of network manager module **1402**. If the data is held, routine NM_ReleaseData is called after network data has been processed to release the data.

Routines TID NM_StartData, NM_EndData, NM_SetDataLength, NM_GrowDataLength, NM_GetDataLength, NM_GetDataPointer, and NM_DeliverData are used internally by network manager module **1402**, and by local services to deliver data. By allowing local services to use these routines, the same buffers can be used to store both network and locally generated data thereby reducing the amount of memory required to support client module **702**.

Routine TID NM_StartData creates a new data transaction and triggers a data delivery event. Routine NM_EndData is called when all data for the given server transaction identification code TRANSACTIONId has been transmitted. Routine NM_SetDataLength sets the data segment to a given length and may cause the location of the data to change. Routine NM_GrowDataLength grows the data segment by a given length and also may cause the location of the data to change. Routine NM_GetDataLength returns the length of the data segment. Routine NM_GetDataPointer returns a pointer to the data. This routine is preferably called before writing into the data buffer. Also, this routine is preferably called whenever the data's location may have changed. Routine NM_DeliverData can be called when at least one card has been stored to reduce latency while the other cards are being generated.

TIL manager module **1403** insulates the rest of client module **702** from changes to the TIL specification. The interface provided by TIL manager module **1403** has the following characteristics: removes the need for parsing by the rest of client module **702**; uses cursors to avoid generating data structures on-the-fly; does not need an entire deck to operate; and handles TIL versioning.

Each TIL deck contains a major and a minor version number. The minor version number is incremented when TIL changes in a way that does not break existing TIL manager modules. The major version number is incremented for non-compatible versions of TIL.

Each TIL deck has the same hierarchy. One embodiment of this hierarchy is presented in Table 6. In Table 6, indentation is used to represent the relationships of the various hierarchical levels.

TABLE 6

TIL DECK HIERARCHY

```
deck
    options
    softkeys
        options
    card
        options
        softkeys
            options
        formatted text
            formatted lines
        entries
            options
formatted line
```

The interface presented in Table 7 for TIL manager module **1403** is designed with the assumption that TIL is a direct tokenization of PIDL as described in Appendix I. However, the interface does not have any dependencies on that tokenization and can support other PIDL encoding techniques. Given the above assumption, the opaque pointers described below are actual pointers into the TIL deck itself. A rudimentary object typing scheme based on where in the deck the opaque pointer points can be used to implement the generic functions described below. If this object typing is not feasible due to details of TIL encoding, the generic functions can be replaced with specific functions.

TABLE 7

ARCHITECTURE FOR TIL MANAGER MODULE 1403

```
        typedef char *opaque;
        typedef opaque Deck;
        typedef opaque Card;
        typedef opaque Text;
        typedef opaque Entry;
        typedef opaque Option;
        typedef opaque SoftKey;
        typedef opaque Object;
/* Generic functions */
        FirstOption(Object obj, Option *o);
            /* obj is a card, softkey, entry, or deck */
        GetSoftkey(Object obj, Option *o);
            /* obj is a card or deck */
        GetText(Object obj, Option *o);
            /* obj is a card or entry */
/* Deck functions */
        SetDeck(Deck d, int length);
            /* tells module which deck to use */
        DeckGetCard(Card *c, int num);
-or-
        DeckGetCard(Deck d, Card *c, int num);
/* Card functions */
        int CardType(Card c);
        CardFirstEntry(Card c, Entry *e);
        CardLookupSoftkey(Card c, int num, Softkey *s);
        CardIsLast(Card c);
/* Option cursor functions */
        OptionNext(Option *o);
        char *OptionKey(Option o);
        char *OptionValue(Option o)
/* Entry cursor functions */
/* Text (and image) cursor functions */
        TextNextToken(Text *t, int *type, int *subtype,
                                int *length, char *data);
```

Archive manager module **1404** stores and retrieves long-lived information. This information includes: data related to the server's location and/or required to support server scaling; data related to encryption; TIL caching (transparent to user); TIL storage (specified by user); and message storage and retrieval (see local manager module) . Archive manager module **1404** should support a variety of nonvolatile memory schemes that are provided by the two-way data communication devices.

Local manager module **1405** is an interface to local device resources, such as local messages, address book entries, and preconfigured e-mail replies. Local manager module **1405** should also define an abstract interface to navigation manager module **1401** for use by archive manager module **1404**.

Table 8 is an architecture for an interface within local manager module **1405** to access to an address book stored on cellular telephone **700**. The name of a routine in Table 8 is descriptive of the operations performed by the routine.

TABLE 8

ARCHITECTURE FOR ADDRESS BOOK ACCESS

```
int NumAddresses();
char *AddressName(int num);
char *AddressGetEMail(int num);
      // returns e-mail address
char *AddressGetPhone(int num);
      // returns phone number
char * AddressGetFax(int num);
      // returns fax number
SetAddress(int num, char *name, char *email,
            char *phone, char *fax);
DeleteAddress(int num);
InsertAddress(int before);
```

Table 9 is an architecture for an interface within local manager module **1405** to access predetermined replies stored on cellular telephone **700**. The name of a routine in Table 9 is descriptive of the operations performed by the routine.

TABLE 9

ARCHITECTURE FOR PREDETERMINED REPLY ACCESS

```
int NumReplies();
char * GetReply(int num);
DeleteReply(int num);
SetReply(int num, char *text);
InsertReply(int before);
```

Table 10 is an architecture for an interface within local manager module **1405** to access messages stored locally on cellular telephone **700**. The name of a routine in Table 10 is descriptive of the operations performed by the routine.

TABLE 10

ARCHITECTURE FOR LOCALLY STORED MESSAGE ACCESS

```
int NumMessages();
void *FirstMessage();
void *NextMessage();
int MessageType(void *msg);
      // e.g. e-mail, TIL, etc.
void *MessageContent(void *msg);
void *SaveMessage(int type, void *content, int
                  contentLength);
DeleteMessage(void *msg);
```

Event manager module **1406** handles the distribution of events. In this embodiment, events include low-level events like key presses and higher level navigation and user interface events. There are typically only a small number of events at any one time. The main event loop in the two-way data communication device dependent module keeps calling EM_GetNextEvent ( ) until no events are left in the queue. Note that processing one event can cause another event to be pushed onto the queue. The main event loop is not restarted until another event is pushed onto the queue due to a user key press or a network event.

In this embodiment, the event types include:

1) keypad events, i.e., pressing of a key;

2) choice events relating to a current choice card, e.g., the user selecting choice three;

3) text entry events relating to a current entry card, e.g., the user keying in “Hello”;

4) network events , e.g., response arrived, request arrived , transaction terminated, network status; and

5) history events, e.g., pop, pop to marker.

Table 11 is an architecture for event manager module **1406**. As in the other tables herein, the name of a routine in Table 11 is descriptive of the operations performed by the routine and in addition a brief description is given in the comment field.

TABLE 11

ARCHITECTURE FOR EVENT MANAGER MODULE 1406

```
struct Event {
    int type;
void *data;
      /* e.g. keycode, choice num, entry
           text, status code, other data */
}
EM_QueueEvent(int type, void * data);
      /* Adds event at end of queue*/
EM_GetNextEvent(Event * event);
      /*Pops next event*/
EM_PeekNextEvent(Event event);
      /*Peeks at next event*/
```

Timer manager module **1407** allows timer events to support timeouts, animation, and other time-domain features. Timeouts are delivered via event manager module **1406**.

Table 12 is an architecture for timer manager module **1407**. As in the other tables herein, the name of a routine in Table 12 is descriptive of the operations performed by the routine.

TABLE 12

ARCHITECTURE FOR TIMER MANAGER MODULE 1407

```
TimerInit();
int TimerSet(int milliseconds, int code, void
                  *clientData);
     /*Returns a timer identification timerId to
          be used for cancellations*/
TimerCancel(int timerId);
TimerCancelAll()
```

User interface manager module **1408** handles interactions with the keypad and the display. Each of the three types of user interfaces defined in Table 1 above requires a different version of user interface manager module **1408**. For most cellular telephones, only one card at a time is used. However, some cellular telephones can display multiple cards at once and so would require a different version of user interface manager module **1408** from the version that handled display of only one card at a time.

In this embodiment, user interface manager module provides a user interface for the three types of cards display, choice, and entry; provides hooks for custom user interfaces for the address list and e-mail reply entry; only cares about the user interface aspects of cards and provides no navigation, argument, or option processing; handles all text and graphic layout including word wrapping; handles scrolling of text; operates from PIDL data structures; generates keyboard events, some of which may be generated by soft keys; and generates high-level events, e.g. next card, choice entry 3, text entry “IBM”.

Table 13 is an architecture for processing cards by user interface manager module **1408**. As in the other tables herein, the name of a routine in Table 13 is descriptive of the operations performed by the routine.

TABLE 13

ARCHITECTURE FOR CARD PROCESSING BY UI MANAGER MODULE 1408

```
void UI_StartCard(Card c);
      /* called to begin display and processing of
           a given card*/
void UI_EndCard(Card c);
      /*called when a card is no longer to be
           displayed*/
Boolean UI_HandleEvent(Event *pevent);
      /*returns true if the event is handled, false
           if not*/
```

Table 14 is an architecture for the user interface implementation by user interface manager module **1408**. As in the other tables herein, the name of a routine in Table 14 is descriptive of the operations performed by the routine.

TABLE 14

ARCHITECTURE FOR UI IMPLEMENTATION BY UI MANAGER MODULE 1408

```
UI_ LayoutCard(Card c, Boolean draw, Proc
                                callback)
      /* relies on global data; needs to be able
           to: draw as it goes; and note the
           special function of the currentLine
           (e.q. none, choice, softkey)*/
int numLines, firstVisible, lastVisible,
                                currentLine;
char currentEntry[80];
int currentChoice;
void *currentSoftkey;
Card currentCard; and
      . . . other info as needed for in-line
           scrolling
```

The callback routine is notified of the special function of each line as the line is laid out. Thus, routine UI_LayoutCard can be used to scroll to a particular choice. If the current line is too wide to display all at once, horizontal scrolling is used to display the complete line, one display width at a time.

Memory manager module **1409** is optional, and is used in two-way data communication devices that do not support dynamic memory allocation. In these devices, all memory allocation and releases must go through memory manager module **1409**. Also, by allocating memory in advance via memory manager module **1409**, client module **702** does not run out of memory due to some other process on the device using up memory.

Microfiche Appendix A is a computer source code listing in the C++ computer language of one embodiment of a client module within a cellular telephone, and one embodiment of an airnet network translator that was used with an Internet server to communicate with client module. The Internet server was a UNIX computer running the Mosaic HTTP server. The source code was used to generate executable code by compiling the source code on a computer running the Sun Microsystems Operating System Solaris 2.4 using Sun Microsystems compiler SunPro C and C#, and the Sun Microsystems SDK make utility. All of these products are available from Sun Microsystems of Mountain View, Calif.

This application is related to copending and commonly filed U.S. patent application Ser. No. 08/XXX,XXX entitled "A PREDICTIVE DATA ENTRY METHOD FOR A KEYPAD" of Alain Rossmann, which is incorporated herein by reference in its entirety.

Various embodiments of a novel interactive two-way data communication system, a two-way data communication device, an airnet network architecture, and a predictive text entry system have been described herein. These embodiments are illustrative only of the principles of the invention and are not intended to limit the invention to the specific embodiments described. In view of this disclosure, those skilled in the art will be able to use the principles of this invention in a wide variety of applications to obtain the advantages of this invention, as described above.

APPENDIX I

A DESCRIPTION OF PIDL AND TIL

The main structure of PIDL is described by an abstract syntax. This appendix describes the elements of the language and their semantics. In the syntax description of each element, an element is defined in 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	the element b followed by element c, the intervening space is just for clarity
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 topions of the element a, see Options below

In general, the element blank-space can optionally appear between any two other elements. To keep the diagram clear, it has been omitted except where required. Where a blank-space is illegal or treated specially, it is noted.

L:\DMS\7574\M-3423_U\0135864.04A

The PIDL ELEMENTS

```
deck          ::= deck-header {softkey}* {card}+ deck-
                  footer
deck-header   ::= <PIDL  ol(deck-options) >
deck-options  ::= o-args | o-cost | o-ttl
deck-footer   ::= </PIDL>
```

A deck consists of one or more cards. There must be at least one card. A deck may also have a number of softkeys defined that stay in force for the whole deck. See soft keys below for the syntax and full description.

```
o-cost        ::= cost=  value
o-ttl         ::= ttl=  integer
```

Additional arguments to be passed on the next deck request are given in o-args. See Arguments below for syntax and full description.

The cost of retrieving this page (exclusive of telephone system charges) is represented in o-cost. If no o-cost is given, the deck cost is included with the user's standard service contract.

Decks can be cached by the cellular telephone for a period of time. The o-ttl entry indicates the number of seconds that the deck can be cached from time of reception. If no o-ttl entry is given, the deck can only be cached for short periods of time, for example, to

implement a back function similar to that of most Web browsers. If the value of o-ttl is zero, the deck must not be cached.

CARD ELEMENTS

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

A card is one of three types of card in this embodiment. These are described in the sections below.

card-options ::= o-name | o-next | o-prev

o-name ::= **name=** identifier

o-next ::= **next=** destination

o-prev ::= **prev=** destination

All cards can have these options. The optional o-name option gives a name to the card. If a card has a name, the card can be referred to by a destination.

The o-next and o-prev give destinations for the NEXT and PREV keys. If omitted, the defaults are the next and previous sequential card in the deck. If o-prev is omitted from the first card, the PREV key returns to the deck last visited. If o-next is omitted from the last card, the NEXT key returns to the first card of the current deck. However, this default behavior is only a fail-safe: the last card in a deck should always have either an o-next option, or be a choice card where each choice entry indicates a new destination.

DISPLAY CARD

```
          display-card  ::= display-header display-content
                        display-footer
        display-header  ::= <DISPLAY option-list >
       display-options  ::= card-options
       display-content  ::= { softkey }* 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.
                        Softkeys can be described for this card
                        only, see Softkeys below.

                          CHOICE CARD
          choice-card   ::= choice-header display-content
                        { entries } choice-footer
        choice-header   ::= <CHOICE  ol{choice-options} >
       choice-options   ::= card-options | o-method | o-key
                        | o-default
            o-method    ::= method= method-type
          method-type   ::= number | list | alpha | group
              entries   ::= { choice-entry }+
                        ::= { group-entry { choice-entry }+ )+
        choice-footer   ::= </CHOICE>

                        Choices let user pick one from a list.
                        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 or scrolled
                        by the phone if too long).

                        How the choices are displayed and chosen
```

is based on the o-method option. Note that this option is a hint only, and can be disregarded by the phone. The number method is the default and indicates that the choices are numbered sequentially from one and are chosen by pressing the appropriate digit on the keypad. If there are more than nine options, the phone may choose some other method of selection. The list method indicates that the list should be unnumbered and that the user should scroll through the list and hit some designated enter key to choose an entry. The alpha method is like list, only it is an indication that the text of the entries should be used to aid selection if at all possible. In this case, the entries are assumed to be alphabetically sorted. The group method is described in more detail below.

The o-key option indicates, if present, the key of an argument to be added to the argument list. See Arguments below for more information. The value of the argument comes from the choice entry; see below. The o-default option indicates the default value if the user just hit ENTER. See o-default under Entry Card below for more information.

choice-entry ::= **<CE** ol(entry-options) **>** formatted-line

entry-options ::= action-options | o-value

Each choice has text displayed to the user. If the action-options are given,

the indicated action is performed if the choice is made. If the o-value option is present, it supplies the value to the argument identified with the o-key option in the choice header. If no o-value is given, the text of the entry is used (without any formatting) as the argument value.

group-entry ::= **<GE** ol(group-options) **>**

group-options ::= **label=** value

If the group method is used, the choices are divided into a number of groups. Each group is headed by a group-entry, which, via the label option, gives a short name to the group. The phone can then give the user a hierarchial interface for choosing among a large number of choices. The text of the label should be limited to eight characters and may be truncated by the phone.

ENTRY CARD

entry-card ::= entry-header display-content entry-footer

entry-header ::= **<ENTRY** ol(entry-options) **>**

entry-options ::= card-options | o-format | o-key | o-default

entry-footer ::= **</ENTRY>**

Entries let the user enter a value. The display content is shown to the user, followed by an entry line. The user's entry is controlled by the format. The o-key option indicates the argument that

is being set by this entry. The value of the argument are the user's entry.

o-format ::= **format=** value { ; format-hint }

format-hint ::= value

This 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. Most of the format control characters control what data is expected to be keyed in by the user. They are displayed as blanks until the user types into them.

The format codes are:

A	entry of any alphabetic character
9	entry of any numeric character
X	entry of any alphabetic or numeric character
***f**	allow entry of any number of characters; the next character, **f**, is one of A, 9 or X and specifies what kind of characters can be entered.
\c	display the next character, **c**, in the entry field; allows display of the formatting characters in the entry field.

Format hints indicate what kind of value is expected. If a format hint is not understood, it is ignored. Currently defined format hints are:

text	text is expected to be text, use special input techniques; generally follows ***A** or ***X**
mail-reply	like text, but expected text is for an e-mail message or page; may affect input algorithm
address-list	entry is a list of e-mail addresses

o-default ::= **default=** value

The o-default option supplies a value that is used if the user simply hits NEXT. If no default value is given, then the user must supply a value.

FORMATTED TEXT

formatted-text ::= { { flow-image } { line-format } text-line)*

formatted-line ::= text-line

text-line ::= { text | image | text-format | alignment-format }*

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

text-format ::= **<B>** | **<I>** | **<BL>**

::= **</B>** | **<I>** | **<BL>**

The format codes control Bold, Italic and Blinking. The slash versions cancel the formatting. Unlike HTML, these needn't be strictly nested and over application and over cancellation are

tolerated. Formatted-text and formatted-line elements start in plain mode (no bold, italic, or blinking).

alignment-format ::= **<CENTER>** | **<BOLD>** | **<TAB>**
The alignment codes specify how parts of a line are to be laid out. The text following the alignment code is either centered or right justified on the same line as the other text. The text or image following the code is considered to be all text up to the next alignment code or line break. All lines start implicitly aligned left. Note that these do not include an implicit line break so that one can have both left and right justified text on a single line. If there is too much text and not enough room on the line then, if in wrap mode, the non-fitting text is moved to the next line and aligned the same way. If in line mode, the line may end up running together with two spaces between the left, center, and right justified segments.

The tab code is used to create aligned columns. Rather than tab to specific character positions, the tab code separates the text for each column. The width of the column is determined by the maximal width of the text (or images) in each line. The extent of the columns is from the first line with tab codes through the last contiguous line with tab codes. Some lines may have fewer

tab codes than others, in which case they are assumed to have no text for the extra columns.

line-format ::= **<WRAP>** | **<LINE>**
::= **
**

Multiple lines of text are separated by the
 code. If a line is too long to fit on the screen and, if in wrap mode, the line is word wrapped onto multiple lines. If in line mode, the line is left as one line and is scrolled horizontally. Formatted-text and formatted-line elements start in wrap mode and may be changed with either the <WRAP> or <LINE> codes. These codes are an implicit line break.

IMAGES

image ::= **<IMAGE** ol(image-options) **>**
::= **<INLINE** ol(image-options) **>**
inline-data **</INLINE>**

flow-image ::= **<IMAGE** ol(flow-image-options) **>**
::= **<INLINE** ol(flow-image-options) **>**
inline-data **</INLINE>**

image-options ::= o-source

flow-image-options ::= image-options | o-flow

inline-data ::= *ASCII85 encoding of image data*

Images are treated as large words and, by default, are simply displayed as part of the text. Flow-Images have a flow option that causes them to be treated differently. The image data is stored in a separate data stream as identified

L:\DMS\7574\M-3423_U\0135864.04A

by the source option.

Inline images are treated identically, only the data is part of the current data stream. ASCII85 is a standard way of encoding binary data in printable ASCII, whereby each four bytes of data is encoded in five characters. Note that TIL only uses inline images, and uses a different encoding.

o-source ::= **src=** location
This option specifies the location of the source for images.

o-flow ::= **flow=** { **left** | **right** }
This option controls the alignment of flow-images. The option specifies that the image is flush left or flush right with the screen. Subsequent lines of text flow in the remaining right or left hand space.

SOFTKEYS

softkey ::= **<SOFTKEY** ol(softkey-options) **>**
softkey-options ::= o-label | o-button | action-options

Softkeys supply definitions for two buttons known as SOFT-L and SOFT-R. They do not show up in the normal text and graphics area displayed to the user, but on a separate line for soft key labels. (Note: in some implementations, where screen real estate is scarce, this label line may get used for normal text

and graphics display when there are no softkeys defined on the current card).

When the softkey is pressed, the indicated action takes place.

o-button ::= **button=** side
side ::= **left** | **right**
o-label ::= **label=** value

The button option specifies which physical key the softkey applies to. The label option is the text that is displayed on screen for that key. The phone may truncate the label. It is suggested that labels be fewer than eight characters.

Softkeys can be specified both for the deck as a whole and per card. When specified for the deck (after the deck-header, but before the first card) they remain in effect for the entire deck. When specified for a card (at the beginning of the formatted-text for the card), they temporarily override any deck softkeys while the card is visible. Note that the override is done independently for the two keys (a card can override one softkey, but not the other). To override a deck softkey with no softkey (in effect, to remove a softkey for the duration of a card) use a softkey with no label and no action.

SYNTAX: OPTIONS

Many of the syntactic elements of PIDL have option lists associated with them. Options refine the operation of the elements they are part of. Unless otherwise noted, options do not nest, even when the same option is given in two nested elements. Options that are not defined for an element are ignored, even if valid for an enclosing element.

```
ol(valid-option)  ::= { blank-space valid-option }*
                      {blank-spaces }
     option-list  ::= ol(option)
          option  ::= key = value
             key  ::= identifier
           value  ::= plain-text
                  ::= " { text } "
```

An option list contains zero or more options. Each option is separated by blank-space (required!) 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. They may be given in any order within the list of options. The key is always an alpha-numeric name that is case insensitive. The value, if it is composed of only alphanumeric characters, may appear directly after the equals sign. Otherwise, the value

must be quoted. In quotes, blank-space 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.

Unlike almost all other syntactic elements, blank space is **not** permitted between the key and the equals sign or between the equals sign and the value.

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

DESTINATIONS

```
destination  ::= location { ; animation }
             ::= card-loc { ; animation }
             ::= stack-operation { ; animation }

   location  ::= full-loc  | partial-loc
             | relative-loc
   full-loc  ::= : service-id / deck-path
             ::= : service-host / deck-path
partial-loc  ::= / deck-path
relative-loc { ../ }* deck-path

   card-loc  ::= # identifier

  deck-path  ::= plain-text { / plain-text }*
 service-id  ::= % plain-text
```

service-host ::= plain-text

Destinations are used in some options to indicate the next, or previous deck or card to show. A deck is specified either with a full location (service-id and deck-path), just a deck-path (in which case the service is the same as the current deck's service), or a relative deck-path. In the later case, the last component of the current deck-path is removed, (and one additional component for each ../ in the relative deck-path), and the deck-path appended.

A particular card can be a destination and is specified by a card-loc element.

stack-operation ::= **+** card-loc

::= **-**

In addition to the normal history list of where a user has been that is kept by a phone, the phone also keeps a short stack of locations. Using a plus sign form causes the current location (deck and card, and location in the history list, and animation used) to be pushed on the stack before going to the new card. Using the minus sign form causes a return to the location on the top of the stack, and the history list to be pruned back to the saved point. If no animation is given, the inverse animation is used. The stack is popped.

animation ::= **slideN | slideS**

L:\DMS\7574\M-3423_U\0135864.04A

::= **slideW** | **slideE**
::= **slideSW** | **slideNE**
::= **slideSE** | **slideNW**
::= **flipV**
::= **flipH**
::= **fade**
::= **none**

The optional animation argument indicates what form of screen animation, if available, is to be used when going to the destination. The animation is remembered with the destination in the history and destination stack. If the user moves to a destination via a 'go' or 'next' operation, then the animation is performed. If the user moves to a destination via a 'prev' or 'pop' operation, the reverse animation associated with the current location is performed.

ACTIONS

Choice entries and soft keys can specify actions to be performed when the user selects the choice or softkey.

action-options ::= o-args | o-call | o-page

o-go ::= **go=** destination

o-call ::= **call=** value

The go operation indicates that the destination should be moved.

ARGUMENT PROCESSING

Each time a deck is requested, arguments may be passed along with the request.

These arguments may be used by the service end to compute a deck specific for the user rather than just return a pre-written deck.

Arguments are built-up as the user traverses the deck. Each argument is a key-value pair. While arguments superficially look like options, these two entities are quite distinct: Options are a part of PIDL and affect the operation of the phone. Arguments are information gathered by the phone and returned to the service. Neither PIDL nor the phone understands the arguments beyond their basic syntactic structure.

Arguments come from three places: Choice cards, Entry cards, and the args option. Each of these specifies a key-value pair that is added to a buffer of arguments to be sent. In the case of the args option, multiple arguments may be specified. When an argument key-value pair is added to the argument buffer, if the key is already present in the buffer, its value is replaced.

```
          o-args  ::= args=  arg-list
argument-list  ::= arg-key-value { { &  |  ; }
                   key-value }*
arg-key-value  ::= arg-key = arg-value
       arg-key  ::= identifier
     arg-value  ::= plain-text
```

Note: The entire o-args element is

actually the value of an option. If it has more than one arg-key-value it will need to be in quotes. Since the ampersand (&) and semi-colon (;) are used as key-value pair separators, these characters cannot be part of argument values.

o-key ::= **key=** arg-key
o-value ::= **value=** arg-value
These options are used in choice and entry cards to specify the key and value for the arguments those cards set.

BASIC ELEMENTS

alpha ::= *any alphabetic character*
numeric ::= *any digit*
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 7-bit ASCII character except <, >, ", or &*
::= **>** | **<** | **"** | **&**

L:\DMS\7574\M-3423_U\0135864.04A

```
                    |   
                    ::= any ISO-Latin-1 named entity
                    ::= &# hex hex ;
                    In text, runs of blank-space are treated
                    as single spaces and my be used as point
                    for word wrapping.
          plain-text
               safe ::= { alpha  |  numeric  |  safe }*
                    ::= $  |  -  | _  | @  |  .  |  &  |  !
                    |  *  |  ,
```

L:\DMS\7574\M-3423_U\0135864.04A

APPENDIX II

A COMPUTER PROGRAM TO GENERATE

A LETTER FREQUENCY TABLE

FOR USE IN THE PREDICTIVE DATA ENTRY PROCESS


```
/*   This program opens a text file selected by the
     user, generates the frequency table for that file,
     and then writes the frequency table to another
     file also selected by the user.
*/

#include <stdio.h>
#include <string.h>
#include <console.h>
#include <assert.h>


typedef unsigned char byte;
typedef byte triplet[3];
typedef byte tristorage[27][27][27];

IncrementTrigram(triplet t, tristorage trigrams)
{
  byte * pb;
  assert(t[0] < 27);
  assert(t[1] < 27);
  assert(t[2] < 27);
  pb = &trigrams[t[0]][t[1]][t[2]];
  if (*pb < 255) *pb = *pb + 1;
  return *pb;
}

StoreTrigramValue(triplet t, tristorage trigrams, byte
value)
{
```

<PIDL>	90	args=	C0	alpha	EO
</PIDL>	91	button=	C1	center	E1
<DISPLAY>	92	call=	C2	fade	E2
</DISPLAY>	93	cost=	C3	flipH	E3
<CHOICE>	94	default=	C4	flipV	E4
</CHOICE>	95	flow=	C5	group	E5
<ENTRY>	96	format=	C6	inline	E6
</ENTRY>	97	go=	C7	left	E7
<CE	A0	key=	C8	list	E8
<GE	A1	label=	C9	none	E9
<IMAGE	A2	method=	CA	number	EA
<INLINE	A3	name=	CB	right	EB
<SOFTKEY	A4	next=	CC	slideE	EC
<B>	B0	page=	CD	slideN	ED

-continued

</B>	B1	prev=	CE	slideNE	EE
<I>	B2	src=	CF	slideNW	EF
</I>	B3	ttl=	D0	slideS	F0
<BL>	B4	value=	D1	slideSE	F1
</BL>	B5			slideSW	F2
<CENTER>	B6			slideW	F3
<RIGHT>	B7				
<WRAP>	B8				
<LINE>	B9				
 	BA				

L:\DMS\7574\M-3423_U\0135864.04A

```
    assert(t[0] < 27);
    assert(t[1] < 27);
    assert(t[2] < 27);
    trigrams[t[0]][t[1]][t[2]] = value;
}

byte FetchTrigramvalue(triplet t, tristorage trigrams)
{
    assert(t[0] < 27); .
    assert(t[1] < 27);
    assert(t[2] < 27);
    return trigrams[t[0]][t[1]][t[2]];
}

byte DumpTrigram(triplet t, tristorage trigrams)
{
    byte value;
    assert(t[0] < 27);
    assert(t[1] < 27);
    assert(t[2] < 27);
    value = FetchTrigramValue(t, trigrams);
    if (value != 0)
    {
      printf("%c%c%c = ", t[0] + 'a', t[1] + 'a', t[2] +
'a');
      if (value == 255) printf("***");
      else          printf("%3d", value);
}
 return value;
 }

 int IdFromChar(short c)
 {
   c = tolower(c);
   if (c < 'a' || c > 'z') return 26;
   return c - 'a';
```

```
}

AddChar(tristorage trigrams, triplet t, byte b)
{
  byte value;
  unsigned short r;

  assert(b <= 26);
  if (b == 26) { t[0] = t[1] = t[2] = 26; return; }

  t[0] = t[1];
  t[1] = t[2];
  t[2] = b;

  value = FetchTrigramValue(t, trigrams);
  if (value == 255) return;

#if 0
  if (value > 64) {
    r = Random();
    if (value > 192 && r & OxE000) return;
    else if (value > 128 && r & OxC000) return;
    else if (value > 64 && r & Ox8000) return;
  }
#endif

  StoreTrigramValue(t, trigrams, value + 1);
}

DumpTrigrams(tristorage trigrams)
{
  int i, j, k;
  int x;
  triplet t;
  x = O;
  for (i = 0; i < 26; ++i)
```

```
        for (j = 0; j < 26; ++j)
           for (k = 0; k < 26; ++k)
           {
              byte value;
              t[0] = i;
              t[1] = j;
              t[2] = k;


              value = DumpTrigram(t, trigrams);

              if (value == 0) continue;
              if (++x == 6) {
                 printf("\n"); x = 0;
              }
              else
                 printf(" ");
           }
     }

     OSErr BuildTrigram(short refNum, tristorage trigrams)
     {
        OSErr err;
        triplet t;
        t[0] = t[1] = t[2] = 26;

        while (true)
        {
           long count = 80;
           char buf[80];
           int i;

           err = FSRead(refNum, &count, buf);
           if (count == 0) return err;
           for (i = 0; i < count; ++i) {
              AddChar(trigrams, t, IdFromChar(buf[i]));
```

```
        }
        if (err) return err;
      }
      return 0;
}

Handle OpenTrigrams(void)
{
      OSErr err;
      OSType type;
      StandardFileReply reply
      short refNum;
      short id;
      Handle h;
      Str63   name;
      tristorage *trigrams;

      type = 'TEXT';
      StandardGetFile(nil, 1, &type, &reply);
      if (!reply.sfGood)   return nil;
      err = FSpOpenDF(&reply.sfFile, fsCurPerm, &refNum);
      if (err) return nil;

      memcpy(name, reply.sfFile.name, sizeof(name));

      h = NewHandle(sizeof(tristorage));


      HLock(h);
      trigrams = (tristorage *)(*h);
      memset(*trigrams, 0, sizeof(tristorage));

      BuildTrigram(refNum, *trigrams);
      FSClose(refNum);
      DumpTrigrams(*trigrams);
      HUnlock(h);
```

```
    type = 'rsrc';
    StandardGetFile(nil, 1, &type, &reply);
    if (!reply.sfGood) return;
    refNum = FSpOpenResFile(&reply.sfFile, fsCurPerm);
    if (refNum == -1) return;
    UseResFile(refNum);

    id = UniquelID('smrt');
    //id = 128;
    AddResource(h, 'smrt', id, name);
    UpdateResFile(refNum);
    FSClose(refNum);
    return h;
}

main()
{
    OSErr   err;
    Handle    h;

    cshow(stdout);
    TEInit();
    InitDialogs(OL);
    InitCursor();
    h = OpenTrigrams();
```

I claim:

1. A two-way data communication system for communication between a computer and a two-way data communication device selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, said two-way data communication system comprising:

a two-way data communication network;

a server computer comprising:

a two-way data communication interface module coupled to said two-way data communication network; and

a server coupled to said two-way data communication interface module;

wherein said server receives a message including a resource locator from said two-way data communication network, and said resource locator includes an address of said server;

said server processes said message using said resource locator; and

said server transmits a response to said message over said two-way data communication network;

a two-way data communication device coupled to said two-way data communication network wherein said two-way data communication device is selected from the group consisting of a cellular telephone, a two-way pager, and a telephone, and further wherein said two-way data communication device further comprises:

a network interface module coupled to said two-way data communication network; and

an client module coupled to said network interface module;

wherein said client module transmits said message including said resource locator to said server over said two-way data communication network; and

said client module processes said response to said message from said server wherein said response includes information for user interaction over said two-way data communication network.

2. A two-way data communication system as in claim 1 wherein said client module further comprises an interpreter wherein said interpreter generates a user interface using information in said response, and said user interface includes at least one user data input option associated with a resource locator.

3. A two-way data communication system as in claim 2 wherein said resource locator associated with said at least one user data input option addresses an object on said server computer.

4. A two-way data communication system as in claim 2 wherein said resource locator associated with said at least one user data input option addresses an object on another server computer coupled to said two-way data communication network.

5. A two-way data communication system as in claim 1 wherein said interpreter includes a plurality of managers including a user interface manager coupled to a display of said two-way data communication device wherein said user interface manager handles interactions with said display.

6. A two-way data communication system as in claim 5 wherein said user interface manager is coupled to a keypad of said two-way data communication device and further wherein said user interface manager handles interactions with said keypad.

7. A two-way data communication system as in claim 6 wherein upon input of data from said keypad, said interpreter generates another message including another resource locator wherein said another resource locator includes said address of said server and said input data.

8. A two-way data communication system as in claim 7 wherein said another resource locator including said address of said server and said input data comprises a uniform resource locator.

9. A two-way data communication system as in claim 1 wherein said response includes a plurality of resource locators and at least one of said plurality of resource locators includes an address to another server coupled to said communication network.

10. A two-way data communication system as in claim 1 wherein said server is a stateless server and upon said server completing transmission of said response, said server completes all processing of said request and retains no state information for said response.

11. A two-way data communication system as in claim 1 wherein upon said server completing transmission of said response, said server maintains state information concerning said message wherein said server utilizes said state information concerning said message in response to another message from said two-way data communication device.

12. A two-way data communication system as in claim 1 wherein said two-way data communication device further comprises:

a memory; and

a resource locator stored in said memory.

13. A two-way data communication system as in claim 1 wherein said server computer further comprises:

a memory; and

at least one common gateway interface program stored in said memory.

14. A two-way data communication system as in claim 1 wherein said server computer further comprises:

a memory; and

at least one card deck stored in said memory.

15. A two-way data communication system as in claim 14 wherein said at least one card deck includes a display card.

16. A two-way data communication system as in claim 14 wherein said at least one card deck includes a choice card.

17. A two-way data communication system as in claim 14 wherein said at least one card deck includes an entry card.

18. A two-way data communication system as in claim 1 wherein said client module further comprises a predictive text entry module.

19. A two-way data communication system as in claim 1 wherein said two-way data communication device further comprises:

a keypad having a plurality of keys; and

a keypad module coupled to said keypad and to said client module

wherein upon a user pressing a key in said plurality of keys, said keypad module stores information identifying the pressed key in a memory; and

said keypad module notifies said client module of said key press.

20. A two-way data communication system as in claim 19 wherein said client module further comprises a predictive data entry module, wherein said client module uses said predictive data entry module to process said stored information identifying the pressed key upon said client module receiving said notification of said key press.

21. A two-way data communication system as in claim 19 wherein said two-way data communication device further comprises:

a card deck stored in said memory of said two-way data communication device.

22. A two-way data communication system as in claim 21 wherein said at least one card deck includes a display card.

23. A two-way data communication system as in claim 21 wherein said at least one card deck includes a choice card.

24. A two-way data communication system as in claim 21 wherein said at least one card deck includes an entry card.

25. A two-way data communication system as in claim 1 wherein said two-way data communication device further comprises:

a display; and

a display module coupled to said display and to said client module wherein said display module drives said display in response to user interface information from said client module.

26. A two-way data communication system as in claim 19 wherein said two-way data communication device further comprises:

a display; and

a display module coupled to said display and to said client module wherein said display module drives said display in response to user interface information from said client module.

27. A two-way data communication system as in claim 1 wherein said two-way data communication device is said cellular telephone.

28. A two-way data communication system as in claim 1 wherein said two-way data communication device is said two-way pager.

29. A two-way data communication system as in claim 1 wherein said two-way data communication device is said telephone.

30. A two-way data communication system for communication between a server computer and a cellular telephone, said two-way data communication system comprising:

a data capable cellular telephone communication network;

a server computer comprising:

a two-way data communication interface module coupled to said data capable cellular telephone communication network; and

a server coupled to said two-way data communication interface module;

wherein said server receives a message including a resource locator from said data capable cellular telephone communication network wherein said resource locator includes an address of said server;

said server processes said message using said resource locator; and

said server transmits a response to said message over said data capable cellular telephone communication network;

a cellular telephone coupled to said data capable cellular telephone communication network wherein said cellular telephone further comprises:

a network interface module coupled to said data capable cellular telephone communication network; and

an client module coupled to said network interface module;

wherein said client module transmits said message including said resource locator to said server over said data capable cellular telephone communication network; and

said client module processes said response to said message from said server wherein said response includes information for user interaction over said data capable cellular telephone communication network.

31. A two-way data communication system as in claim 30 wherein said client module further comprises an interpreter wherein said interpreter generates a user interface using information in said response and further wherein said interface includes at least one user data input option associated with a resource is locator.

32. A two-way data communication system as in claim 31 wherein said resource locator associated with said user data input option addresses an object on said server computer.

33. A two-way data communication system as in claim 31 wherein said resource locator associated with said user data input option addresses an object on another server computer coupled to said data capable cellular telephone communication network.

34. A two-way data communication system as in claim 30 wherein said interpreter includes a plurality of managers including a user interface manager coupled to a display of said cellular telephone wherein said user interface manager handles interactions with said display.

35. A two-way data communication system as in claim 34 wherein said user interface manager is coupled to a keypad of said cellular telephone and further wherein said user interface manager handles interactions with said keypad.

36. A two-way data communication system as in claim 35 wherein upon input of data from said keypad, said interpreter generates another message including another resource locator wherein said another resource locator includes said address of said server and said input data.

37. A two-way data communication system as in claim 36 wherein said another resource locator including said address of said server and said input data comprises a uniform resource locator.

38. A two-way data communication system as in claim 30 wherein said response includes a plurality of resource locators and at least one of said plurality of resource locators includes an address to another server coupled to said communication network.

39. A two-way data communication system as in claim 30 wherein said server is a stateless server and upon said server completing transmission of said response, said server completes all processing of said request and retains no state information for said response.

40. A two-way data communication system as in claim 30 wherein upon said server completing transmission of said response, said server maintains state information concerning said message wherein said server utilizes said state information concerning said message in response to another message from said cellular telephone.

41. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer comprising:

generating a message by a client module in response to data entered by said user of a two-way data communication device coupled to a two-way data communication network,

wherein said client module executes on a microcontroller of said two-way data communication device;

said message includes a resource locator; and

said two-way data communication device is selected from a group consisting of a cellular telephone, a two-way pager, and a telephone

transmitting said message over said two-way data communication network to a server computer wherein said server computer is identified by said resource locator;

executing an application on said server computer identified by said resource locator to generate a response to said message; and

transmitting said response to a location identified by said application.

42. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **41** wherein said response is transmitted to said client module.

43. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **42** further comprising:

interpreting said response by said client module and generating a user interface using information in said response wherein said interface includes at least one user data input option associated with a resource locator.

44. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **43** wherein said resource locator associated with said user data input option addresses an object on said server computer.

45. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **43** wherein said resource locator associated with said user data input option addresses an object on another server computer.

46. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **43** further comprising:

interpreting a data input entry by a user of said two-way data communication device.

47. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **46** further comprising:

appending said data input entry to said resource locator associated with said data input entry option.

48. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **42** wherein said response is a card deck and further wherein said card deck includes at least one card.

49. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **42** further comprising:

storing said card deck stored in a memory of two-way communication device.

50. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **49** further comprising:

processing said stored card deck using said client module.

51. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **50** further comprising:

generating a display on two-way data communication device for each card in said card deck.

52. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **51** wherein said at least one card is a display card.

53. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **51** wherein said at least one card is an entry card.

54. A method for using a two-way data communication device, selected from a group consisting of a cellular telephone, a two-way pager, and a telephone, to communicate with a server computer as in claim **51** wherein said at least one card is a choice card.

* * * * *