(54) Title: VOICE RESPONSE SYSTEM WITH PROGRAMMING LANGUAGE EXTENSION

(57) Abstract

A telephony voice response system includes a database language sequencer (206-N), a database control module (208-N) having a plurality of procedures callable by the database language for performing database operations, and a telephony control module (210-N) having a plurality of procedures callable by the database language sequencer for performing telephony operations. The telephony operations include spoken predefined prompts onto a telephony channel, receiving and storing DTMF-encoded input from the telephony channel, and recording audio input from the telephony channel. The database language sequencer calls the database control module procedures and the telephony control module procedures in a sequence defined by a program prepared according to a database language. The telephony voice response system can control multiple telephony channels by running a separate task (204-N) for each such channel under a multitasking operating system. A common channel server task is provided which manages the resources of the telephony card for all of the individual channel tasks.
FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

<table>
<thead>
<tr>
<th>Code</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Austria</td>
</tr>
<tr>
<td>AU</td>
<td>Australia</td>
</tr>
<tr>
<td>BB</td>
<td>Barbados</td>
</tr>
<tr>
<td>BE</td>
<td>Belgium</td>
</tr>
<tr>
<td>BF</td>
<td>Burkina Faso</td>
</tr>
<tr>
<td>BG</td>
<td>Bulgaria</td>
</tr>
<tr>
<td>BJ</td>
<td>Benin</td>
</tr>
<tr>
<td>BR</td>
<td>Brazil</td>
</tr>
<tr>
<td>BV</td>
<td>Belarus</td>
</tr>
<tr>
<td>CA</td>
<td>Canada</td>
</tr>
<tr>
<td>CF</td>
<td>Central African Republic</td>
</tr>
<tr>
<td>CG</td>
<td>Congo</td>
</tr>
<tr>
<td>CH</td>
<td>Switzerland</td>
</tr>
<tr>
<td>CI</td>
<td>Côte d'Ivoire</td>
</tr>
<tr>
<td>CM</td>
<td>Cameroon</td>
</tr>
<tr>
<td>CN</td>
<td>China</td>
</tr>
<tr>
<td>CS</td>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>CZ</td>
<td>Czech Republic</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
</tr>
<tr>
<td>ES</td>
<td>Spain</td>
</tr>
<tr>
<td>FI</td>
<td>Finland</td>
</tr>
<tr>
<td>FR</td>
<td>France</td>
</tr>
<tr>
<td>GA</td>
<td>Gabon</td>
</tr>
<tr>
<td>GB</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>GE</td>
<td>Georgia</td>
</tr>
<tr>
<td>GN</td>
<td>Guinea</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
</tr>
<tr>
<td>HU</td>
<td>Hungary</td>
</tr>
<tr>
<td>IE</td>
<td>Ireland</td>
</tr>
<tr>
<td>IT</td>
<td>Italy</td>
</tr>
<tr>
<td>JP</td>
<td>Japan</td>
</tr>
<tr>
<td>KE</td>
<td>Kenya</td>
</tr>
<tr>
<td>KG</td>
<td>Kyrgyzstan</td>
</tr>
<tr>
<td>KP</td>
<td>Democratic People's Republic of Korea</td>
</tr>
<tr>
<td>KR</td>
<td>Republic of Korea</td>
</tr>
<tr>
<td>KZ</td>
<td>Kazakhstan</td>
</tr>
<tr>
<td>LI</td>
<td>Liechtenstein</td>
</tr>
<tr>
<td>LK</td>
<td>Sri Lanka</td>
</tr>
<tr>
<td>LU</td>
<td>Luxembourg</td>
</tr>
<tr>
<td>LV</td>
<td>Latvia</td>
</tr>
<tr>
<td>MC</td>
<td>Monaco</td>
</tr>
<tr>
<td>MD</td>
<td>Republic of Moldova</td>
</tr>
<tr>
<td>MG</td>
<td>Madagascar</td>
</tr>
<tr>
<td>ML</td>
<td>Mali</td>
</tr>
<tr>
<td>MN</td>
<td>Mongolia</td>
</tr>
<tr>
<td>MR</td>
<td>Mauritania</td>
</tr>
<tr>
<td>MW</td>
<td>Malawi</td>
</tr>
<tr>
<td>NE</td>
<td>Niger</td>
</tr>
<tr>
<td>NL</td>
<td>Netherlands</td>
</tr>
<tr>
<td>NO</td>
<td>Norway</td>
</tr>
<tr>
<td>NZ</td>
<td>New Zealand</td>
</tr>
<tr>
<td>PL</td>
<td>Poland</td>
</tr>
<tr>
<td>PT</td>
<td>Portugal</td>
</tr>
<tr>
<td>RO</td>
<td>Romania</td>
</tr>
<tr>
<td>RU</td>
<td>Russian Federation</td>
</tr>
<tr>
<td>SD</td>
<td>Sudan</td>
</tr>
<tr>
<td>SE</td>
<td>Sweden</td>
</tr>
<tr>
<td>SI</td>
<td>Slovenia</td>
</tr>
<tr>
<td>SK</td>
<td>Slovakia</td>
</tr>
<tr>
<td>SN</td>
<td>Senegal</td>
</tr>
<tr>
<td>TD</td>
<td>Chad</td>
</tr>
<tr>
<td>TG</td>
<td>Togo</td>
</tr>
<tr>
<td>TJ</td>
<td>Tajikistan</td>
</tr>
<tr>
<td>TT</td>
<td>Trinidad and Tobago</td>
</tr>
<tr>
<td>UA</td>
<td>Ukraine</td>
</tr>
<tr>
<td>US</td>
<td>United States of America</td>
</tr>
<tr>
<td>UZ</td>
<td>Uzbekistan</td>
</tr>
<tr>
<td>VN</td>
<td>Viet Nam</td>
</tr>
</tbody>
</table>
- 1 -

VOICE RESPONSE SYSTEM WITH
PROGRAMMING LANGUAGE EXTENSION

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure as it appears in the U.S. Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND

1. Field of the Invention
   The invention relates to telephony voice response systems, and more particularly, to the extension of database languages to handle telephony voice response functions.

2. Description of Related Art
   An interactive voice response (IVR) system is a system which allows callers to use a telephone to interact with a remote computer and retrieve data from, or enter data into, one or more databases. Usually callers enter information and commands by pressing buttons on a tone-generating telephone. The telephone generates a DTMF-encoded (dual-tone multi-frequency) signal in response to such buttons, and transmits the tones to the voice response system. The voice response system decodes the tones to determine which buttons were pressed, and proceeds accordingly. In other systems, callers enter information and commands by speaking into the telephone. In such a situation, the voice response system recognizes the words spoken and proceeds accordingly. IVR systems can be as simple as ordinary voice mail systems, or can be highly complex, with
multiple menus and caller-data-entry facilities. They can support either a single telephony channel or multiple simultaneously active telephony channels.

IVR systems are typically developed by programming a general purpose computer system that has telephony hardware installed. For example, IVR systems often include a DOS-based personal computer with a telephony expansion card installed, such as a TyIN 4000 Pro Personal Communication Assistant, available from National Semiconductor Corporation, Santa Clara, California, or a Model D/41 available from Dialogic Corporation, Parsippany, New Jersey.

IVR systems usually need to have a high degree of flexibility for customization by value-added resellers (VARs) and by the MIS departments of end-user customers. VARs will often customize an IVR system for the needs of a particular customer, and many customers need to be able to modify their IVR systems themselves to meet changing requirements for their callers.

In the past, many IVR systems were difficult to customize because they were programmed in an ordinary, general purpose program language, such as C or C++. In order to speed application development and customization, some IVR system suppliers have developed proprietary libraries of C-language procedures which could manage both the control of the telephony hardware and also the data that the caller is accessing. Other suppliers have developed proprietary scripting languages, and provide an interpreter program written in C (or another general purpose programming language). The interpreter follows a script prepared by the developer. Still other suppliers have developed form, table or graphical (GUI) programming environments for IVR system development or customization. VARs and end-
user customers have found all of these mechanisms difficult to learn and use, however, and this has restricted the growth of the IVR market.

One of the problems with the above mechanisms for IVR system development is that while they may be well-suited to managing the telephony aspects of the system, they are not as well suited to managing the database aspects of the system. Database management is best performed by facilities which are designed for that purpose, namely database management systems. A database management system (DBMS) is a software package designed to operate on a collection of one or more computer-stored files, or what is referred to as a database. Its primary operation is to select database records that have user-specified common characteristics, and retrieve those records for further processing and display. The database management system also adds new records to the database, and modifies existing records as desired. A typical database management system can include a non-procedural user interface through which a user at a terminal can cause the DBMS to perform desired operations on the database. A typical DBMS also includes a high-level programming language (referred to herein as a database language or a DBMS language) that can be used procedurally to operate on the data in the database. Often both the non-procedural interface and the procedural interface call a common set of procedures, referred to herein as the database engine, to perform the desired operations on the database. The simple, high-level commands supported by DBMSs to manage a database, such as "seek", "replace", "sort", and so on, are quite powerful. However, DBMSs do not support telephony functions.
It is desirable to use DBMSs in IVR applications also because they inherently manage their databases in the "native format" of the DBMS. Unlike general purpose programming languages which provide enormous flexibility to programmers in the formation of data structures, database management systems organize their databases in a predefined format which users rarely, if ever, need to understand. The high-level commands of a DBMS obviate any necessity for the programmer to be concerned with the underlying format in which the data is actually stored.

Once the data is already maintained in the DBMS native format, a host of additional applications become possible. DBMS language programs can be written easily to operate on the data independently of the telephony connections. For example, a bank might create and manage all of its account information using a DBMS, and have an IVR system for customers to call to retrieve such information. Such an IVR system would need to be able to obtain the desired information from the database in the DBMS native format. As another example, an IVR system might be designed to obtain information from callers and place the information in a database; reports can then be easily generated from the data using the various user interfaces of the DBMS. Thus an IVR system which maintains its data in the native format of a DBMS can be much more tightly integrated with the remainder of the customer's business.

In one conventional attempt to integrate native format database management in IVR systems, the IVR system and a database management system were set up to operate independently on two separate computer systems. The IVR system communicated with the DBMS system via terminal emulation, in which the IVR system acted as a
user terminal communicating and receiving individual characters from a non-procedural terminal interface of the DBMS. The software for the IVR system was written in a general purpose programming language, and the character stream to and from the DBMS system took place through an ordinary I/O port of the IVR system computer. As might be expected, the terminal emulation technique can be extremely slow, inflexible, and arcane.

Another way that IVR system developers have sought to operate on a database in a native DBMS format, as part of an IVR system, was to provide a procedure library, written in a general purpose programming language such as C, which could be compiled with or linked to the main program module of the IVR application, also written in C. However, this technique required a detailed understanding of the DBMS native format and, in large part, duplicated all of the effort that DBMS manufacturers had already expended in the development of their own database engines and tools. It is also rare for IVR system developers to have the expertise necessary to optimize database management software to run as efficiently as that available from the DBMS manufacturer.

Accordingly, as can be seen, prior attempts to integrate IVR systems with native format DBMS databases have left much to be desired. The present invention achieves such integration much more effectively.

SUMMARY OF THE INVENTION

Many existing DBMS languages can be extended using library extension modules. Thus a developer of a database language program can create a proprietary library for operating on the database or for performing
other functions, and can then access the procedures of the library in the same manner as the ordinary procedures of the database engine are accessed. For example, the FoxPro® database management system, available from Microsoft®, includes a "library construction kit" which allows developers to create external libraries of C-language routines that can be integrated into any FoxPro application through a predefined external FoxPro application programming interface (API). Once the library is installed, a FoxPro language program invokes an extension procedure merely by calling it in the same manner that it calls FoxPro built-in procedures. The FoxPro library construction kit is described in Microsoft, "FoxPro® Library Construction Kit, Developer's Guide" (1993), incorporated herein by reference.


These DBMS systems can be thought of as being organized into three components: (1) a database language sequencer module, which follows a database language script (either as an interpreter or as compiled code); (2) a database engine module which contains the DBMS's built-in procedures for operating on the database; and (3) the extension library module, which contains the developer's extension library procedures. The database language sequencer traverses the script, and calls the procedures in either the database engine module or the extension module as required by the script. The interface to these procedures is the same for both the
- 7 -
database engine module and the extension module, as set forth in the above-incorporated references.

Although extension library capabilities have been available in DBMS languages for a long time, they have not heretofore been used for telephony operations. One possible reason for this is that those working in the telephony field developing IVR applications, are used to working in either general purpose programming languages such as C, or in specialized languages developed specifically for telephony operations only.

Accordingly, the invention involves the addition of a telephony library extension to a DBMS that has a programming language, with the resulting combination executing on computer hardware to form a telephony server. In one embodiment, telephony server apparatus includes a database and software instructions executable by a processor structure, the software instructions including a database language sequencer, a database control module having a plurality of procedures callable by the database language for performing at least one database operation, and a telephony control module having a plurality of procedures callable by the database language sequencer for performing at least one telephony operation. The database operations include at least the operations of reading and writing data to a database, and the telephony operations include at least the operations of speaking a predefined prompt onto a telephony channel, receiving and storing DTMF-encoded input from the telephony channel, and recording audio input from the telephony channel. The database language sequencer calls the database control module procedures and the telephony control module procedures in a sequence defined by a program prepared according to the rules of a database language. The database language
sequencer can include either an interpreter or compiled code.

The telephony server apparatus can control multiple telephony channels by running a separate task for each such channel under a multitasking operating system. The above software can be instantiated separately for each task, or parts can be separated out and provided in a common task communicating with the individual channel tasks via inter-process communication (IPC), shared memory, or by another mechanism. In one embodiment, a common channel server task is provided which manages the resources of the telephony card for all of the individual channel tasks.

**BRIEF DESCRIPTION OF THE DRAWINGS**

The invention will be described with respect to particular embodiments thereof, and reference will be made to the drawings, in which:

Fig. 1 is a symbolic block diagram of an IBM PC/AT-compatible personal computer incorporating features of the invention;

Fig. 2 is a block diagram of the software architecture used in the system of Fig. 1;

Fig. 3 is a detail of a sequencer of Fig. 2;

Fig. 4 is a flowchart illustrating the creation of a sequencer in Fig. 2; and

Fig. 5 is another block diagram of the software architecture used in the system of Fig. 1.

**DETAILED DESCRIPTION**

**I. HARDWARE ARCHITECTURE**

Fig. 1 is a symbolic block diagram of an IBM PC/AT-compatible personal computer incorporating features of the invention. It comprises a CPU 102, which may be an
Intel 80486 compatible CPU or an Intel Pentium processor, for example. The CPU 102 has address, data and control lines which are connected to a CPU bus 104. The CPU bus 104 is also connected to a cache memory 106 and to DRAM memory 108, both of which are controlled by system control logic 110. The system control logic 110 is connected to the CPU bus 104 and also to control, address and data lines of an ISA bus 112. Connected to the ISA bus 112 is a ROM 114 containing the system BIOS, a disk controller 116 for floppy and hard-disk drives 118, and one or more telephony cards 120 connected to a plurality of telephony channels 122. The telephony card 120 is a model D/41, available from Dialogic Corporation, Parsippany, New Jersey. In another embodiment, the telephony card is a TyIN 4000 Pro, available from National Semiconductor, Santa Clara, California. The Dialogic board is described in Dialogic, "Voice Hardware Reference" (Dialogic Ref. No. OS-0147-002) (1994), incorporated by reference herein. The TyIN 4000 Pro is described in National Semiconductor, "TyIn 4000 Pro, Getting Started" (1994), incorporated herein by reference. The system of Fig. 1 illustrates only one platform which can run software according to the invention. Numerous other platforms can also suffice, such as Macintosh-based platforms available from Apple Computer, Inc., platforms with different local bus configurations, networked platforms, multi-processor platforms, and so on.

The telephony channels 122 represent separate analog phone lines for each channel. However, a wide variety of other implementations are possible. For example, the telephony channels 122 could represent separate logical channels all carried on one or more telephone-company-provided T1 connections or higher. As another example,
the telephony channels 122 may be carried on one or more ISDN BRI or PRI links into the public switched telephone network. In either case, the telephony cards 120, together with any software drivers, are responsible for presenting the appearance of separate, individual channels (also referred to herein as telephony lines) to higher level software.

Because of the numerous types of hardware platforms which can run software according to the invention, the term "processor structure" as used herein will refer to the CPU or CPUs of single or multiple processor arrangements, whether located in a single box or distributed across a network. Similarly, due to the possibility of paging and overlay mechanisms of different operating systems and application programs as well as memory, disk and network caching, the term "memory structure" includes both volatile and non-volatile memory, mass storage and cache memories, whether these types of memory are all located in a single box or distributed across a network.

II. SOFTWARE ARCHITECTURE

Fig. 2 is a block diagram of the software architecture used in the present embodiment. The software runs under a multitasking operating system such as Microsoft Windows, Windows NT or UNIX. As used herein, the term "multitasking operating system" includes permissive multitasking operating systems as well as preemptive multitasking operating systems. Preferably, the operating system 202 in Fig. 2 is Microsoft Windows. All of the software and data illustrated in Fig. 2 is present in the memory structure of Fig. 1, although because of paging, memory caching and disk caching and other memory management mechanisms,
different parts may at different times be located physically in one or more of the different components of the memory structure.

In the architecture of Fig. 2, each of the telephony channels is associated with a separate concurrent task (also called a concurrent process) running under the operating system 202. The term "concurrent task" does not require that their instructions be executing simultaneously, although this may be possible on a multi-processor system. In Fig. 2, N tasks are shown, 204-1,...204-N (collectively, 204). In one embodiment, these tasks are all created upon initialization of the IVR system, whereas in another embodiment, the tasks are created only as their associated telephony channels become active.

In a multitasking operating system, a task (or process) can be thought of as being divided into two components: software instructions which are executable by the processor structure, and data. Data is further divisible into read-only data and read-write data. When two tasks are running the same software, the read-write data for each task is maintained in a separate region of the memory structure, whereas, depending on the operating system, there may either be a separate copy of the software instructions and read-only data for each task or the different tasks can share the same copy of the software instructions and read-only data. In the present embodiment, a single copy of the software instructions and read-only data is shared. See Pietrek, "Windows Internals," Addison-Wesley, pp. 216-218 (1993). The entire Pietrek text is incorporated by reference herein. Whether or not two tasks running the same program share their software instructions and read-only
data, the program is considered herein to be separately "instantiated" for each of the tasks.

Each of the tasks in Fig. 2 can be thought of as including three modules 206-i, 208-i and 210-i. The module 206-i is a database language sequencer. The sequencer is the portion which the IVR system developer creates or modifies in order to define the caller's experience with the IVR system. Voice prompts are set up in the sequencer module, as are menu hierarchies, actions in response to caller input, and so on. In order to facilitate the extensive data manipulation and data access which sophisticated IVR systems usually require, the IVR system developer creates the sequencer 206-i using a feature-rich database language rather than with a minimal IVR language or general purpose programming language. Any database language can be used in different embodiments of the invention. The embodiment described herein uses FoxPro, but the languages of many other database systems can be used instead, such as dBase, Paradox, Oracle/SQL, Clipper, and so on.

The DBMS-language sequencer 206-i can include either a fully compiled version of the developer's DBMS-language code, or can include the combination of the DBMS-language code in text form plus an interpreter. Other variations are also possible, such as the combination of a tokenized version of the developer's DBMS code plus a traverser for sequencing through the tokens. Fig. 3 illustrates the interpreter variation. As can be seen, the sequencer 206-i includes the FoxPro code 302 in text form as written by a developer, and a FoxPro language interpreter 304 which parses the FoxPro code 302 in order to determine which actions to take. In this variation, the interpreter 304 includes software
instructions executable by the processor structure, which may be shared between tasks, while the FoxPro code 302 is considered read/write data and is not shared between the tasks. However, in another embodiment, the FoxPro code 302 may be shared between tasks. Note that it will typically be desired that most, if not all, of the telephony channels should present the same user experience to callers. In this case, if the FoxPro code 302 is not actually shared, it can at least be identical among all the channels that require the same handling.

Fig. 4 is a flowchart illustrating the creation of the sequencer 206-1 in the variation in which the sequencer constitutes compiled code. In step 402, the developer prepares the FoxPro code in text form. This may be the same code as 302 in Fig. 3. In step 404, the FoxPro code is compiled using the FoxPro DBMS compiler in the FoxPro Distribution Kit, available from Microsoft Corporation, Redmond, WA. This compiler is described in Microsoft, "FoxPro User’s Guide" pp. U6-12 - U6-15 (1993). (The entire User’s Guide is incorporated herein by reference.) In step 406 the compiled sequencer 206-1 has been created.

Since each task 204-1 in Fig. 2 corresponds to a separate telephony channel 122, the IVR system developer can prepare the FoxPro code used to create the sequencer 206-i, as if only one telephony channel was to be handled. The developer need not be concerned with any other telephony channel, or even with the possibility that more than one telephony channel may exist, since all consequences of these possibilities are handled by the multitasking operating system 202 and/or the channel server process 214 described below.

Returning to Fig. 2, the module 208-i represents a database engine for the particular DBMS in which the IVR
system's developer's code was prepared. In the presently described embodiment, database engine 208-i is the FoxPro database engine, available from Microsoft Corporation and incorporated herein by reference.

All of the database engines 208-i contain a set of database management procedures which are callable by the sequencer 206-i in order to manage the data in a common database 212. The database 212 is stored in the memory structure of the computer system of Fig. 1, and for the most part on the disks 118 of such memory structure.

The module 210-i contains a library of telephony-related procedures callable by the sequencer 206-i. The module 210-i can be provided to the IVR system developer in compiled form, so the developer need not be concerned at all with its contents (except to know the identity of, and syntax for, the procedures which may be called from the FoxPro language code). In the situation where the DBMS language used is FoxPro, and the operating system 202 is Microsoft Windows, then the procedures of the module 210-i are preferably written in C and compiled and linked together to create an .FLL (FoxPro Linked Library). The procedures of FLL 210-i are described in more detail below.

The telephony modules 210, each instantiation of which is associated with a different telephony channel 122, communicate with a channel server process 214 which is charged with controlling the telephony cards 120 and allocating their resources in a manner which avoids conflicts among the different tasks 204. The channel server process 214 is described in more detail below as well.

Fig. 5 is another diagram of the software architecture used in the present embodiment. For simplicity, only one task 204 is illustrated, but
greater detail is provided throughout the hierarchy. In addition, since the database engine modules 208 and the database 212 are conventional, they are omitted from Fig. 5.

As can be seen in Fig. 5, the software architecture of the present embodiment is organized into layers. It will be seen that by substituting one module at one layer, different database management systems can be accommodated. By substituting modules at two other layers, different operating systems can be accommodated. Finally, by substituting modules at two further layers, different telephony cards can be accommodated.

Referring to Fig. 5, the top two levels 302 and 304 together form the FoxPro language sequencer 206-1 (Fig. 2). The example of Fig. 3, specifically the combination of FoxPro code in text (layer 302) and a FoxPro language interpreter (layer 304) is illustrated. The FoxPro code in layer 302 is prepared by the IVR system developer, whereas the FoxPro language interpreter 304 is supplied by the DBMS manufacturer (Microsoft, in the case of FoxPro).

The telephony FLL module 210-1 (Fig. 2) comprises three layers: a DBMS interface layer 502, charged with the task of converting parameters between the form in which they are passed to and from the sequencer 206-1, and a generic form used by lower layers in the hierarchy. The layer 502 thus isolates lower layers from having to know which DBMS is running in the higher layers. The module 502 is specific to the particular DBMS used; other versions of this module can be substituted in order to support sequencers from other DBMSs.

Below the DBMS interface layer 502 in the telephony FLL 210-1 is a compute layer 504. Roughly, the compute
layer 504 is charged with performing any computations that are both generic (not specific to a particular DBMS) and not appropriate for the channel server 214 to perform. For example, certain operations should not be performed by the channel server 214 because they may create a bottleneck; these operations are performed in the compute layer 504 instead. Examples of operations performed in the compute layer 504 include retrying an operation if no input is received before a time-out expires; converting a date value input parameter to a list of individual voice prompts to speak; and so on. Different versions of the compute layer 504 may be substituted to meet the needs of different market segments, for example to accommodate different human languages or the customs of different countries.

Below the compute layer 504 in telephony FLL 210-1 is an IPC client layer 506. The IPC client layer 506 communicates with an IPC server layer 508 at the top of the channel server 214 using an inter-process communication (IPC) protocol or mechanism of the underlying operating system 202 (Fig. 2). In Windows implementations, the IPC mechanism can be Dynamic Data Exchange (DDE). Whereas there are N instantiations of the sequencer 206-i and the telephony FLL 210-i, including the IPC client layer 506, there is only one instantiation of the channel server 214, including an IPC server layer 508. The purpose of IPC client layer 506 is to perform the client side of the inter-process communication mechanism for the compute layer 504 in a manner which isolates the compute layer 504 from all details of the IPC communication mechanism. The layer 506 also isolates higher layers from having to know whether the IVR system is serving only a single channel, or multiple channels. The IPC server layer 508 performs
the server side of the inter-process communication mechanism. Note that in systems that support only a single telephony channel, the IPC client layer 506 and the IPC server layer 508 can be combined into a single very simple layer.

Below the IPC server layer 508 in the channel server 214 is a test mode layer 509. When test mode is activated, this layer simulates the operation of the telephony hardware using user dialogs, so that an application can be tested without requiring actual use of telephony hardware. In normal operation, test mode is disabled and the test mode layer 509 simply becomes transparent.

Below the test mode layer 509 is a line driver interface layer 510. The layer 510 implements standard voice/telephony primitives for the particular telephony card(s) installed in the system. In some embodiments, the line driver interface 510 might also control a voice recognition card (not shown). The layer 510 implements such primitives as "answer incoming call", "hang up line", "get DTMF keys", and "play list of prompts". Since the line driver interface layer 510 isolates higher layers from having to know how to control the particular telephony card(s) installed in the system, different versions of the layer 510 may be substituted to handle different telephony cards.

Below the line driver interface layer 510 in the channel server 214 is the telephony card driver 512 supplied by the telephony card manufacturer. For the Dialogic D/41 card, the driver 512 is the D40Drv DOS TSR ("Terminate and Stay Resident") program, available from Dialogic. For the National Semiconductor TyIN 4000 Pro telephony card, the driver layer 512 is the TAPI service provider (TSP) program available from National
Semiconductor. For the Rhetorex telephony card (Model RDSP 9432, available from Rhetorex, Inc., Campbell, CA), the driver 512 is the RhetDrv DOS TSR program available from Rhetorex. All three of these driver programs are incorporated herein by reference in their entirety. Such drivers may also include a number of procedures to be compiled with the channel server 214, which properly call the TSR or TAPI routines.

III. EXAMPLE FOXPRO PROGRAM

Set forth in Appendix A is an example FoxPro program for layer 302 (Fig. 5). The programming language in which the FoxPro code in Appendix A is written, is described in Microsoft, "FoxPro Language Reference" (1993), incorporated herein by reference.

The example program in Appendix A represents the telephony component of an interactive voice response system which schedules part-time workers at a plant. It demonstrates a solution to the need for a large retail outlet to schedule temporary labor, and have temporary employees be able to call in and obtain their work schedule. The voice response system permits workers to enter their worker number, and be told which days of the current week, and which work shift (early or late) they should report to work. They have a chance to accept or reject each individual shift scheduled. As will be seen, the program uses ordinary FoxPro commands (verbs and procedure calls) to access the database. The telephony functions are available by way of new procedure calls, but the syntax for calling these procedures is the same as the ordinary FoxPro language syntax for calling database management procedures. In another embodiment, access to the telephony functions
can be provided by way of newly defined verbs, instead of or in addition to newly defined procedure calls.

The program in Appendix A operates on two databases: a control database called BASEDATA and a working database called WORKERS. The control database contains only two values: a labor rate and an hours-per-shift value. The WORKERS database contains 100 rows, each for a respective temporary worker. The fields in each entry of WORKERS are as follows:

- Worker ID Number 1001-1100
- Address Street
- Address City
- Address State
- Address Zip Code
- Home Phone

Work Schedule Fields es1-es7, ls1-ls7 Boolean values for early shifts 1-7 and late shifts 1-7, indicating whether the worker is scheduled to work the specified shift.

In a portion of FoxPro code prior to that set forth in Appendix A, an initial work schedule was set up in a WORKERS database.

At line 10 of the program, a SET LIBRARY TO command is issued, which loads in a FoxPro linked library called voysaccs. This is the telephony FLL 210-1 (Fig. 5). When FoxPro loads a library, it also runs a start-up procedure of that library which, among other things, registers a procedure for the task 204-1 to handle events in substitution for the default FoxPro event handler.

In line 11, the program calls a telephony FLL function VSET to establish a directory in which
application prompt files are to be found. In line 12, the program calls the same function to establish a directory in which recording files are to be found.

In lines 18-22, certain variables are loaded with base data from the BASEDATA database. In line 28, the actual working database table, WORKERS, is selected. The FoxPro command SET ORDER TO is then used to establish an index based on the field ID.

Lines 36-52 constitute a continuous loop which simply waits for the phone to ring, answers it after one ring, and then calls a procedure pt_call. The function to wait for the phone to ring and answer it after one ring, is VWaitRing(), at line 42, and this is a procedure in the telephony PLL 210-1. The VWaitRing() procedure takes two arguments, both of which are optional. The first argument indicates the number of rings to wait before answering; the default is two. The second argument indicates the number of seconds to wait before timing out; the default is 0, indicating no time-out. Return values indicate success (answered a phone call), time-out, or invalid parameters.

The pt_call function is described below. After pt_call returns, the example program in Appendix A calls another telephony PLL procedure VChkHangup() to determine whether the call is still active (line 47) and, if not, it calls yet another telephony PLL procedure VSpeak() to speak a "good-bye" prompt over the telephony channel. The VSpeak() procedure allows a program to speak a sequence of prompts or values. It can take up to 10 arguments. These arguments can be values (numbers or character strings which contain dates or voice file names), or "speak modes" that regulate how numeric values should be spoken. Speak modes apply only to numeric values, and the default is a standard numeric
format in which, for example, 432 is spoken as "four hundred thirty-two". The speak mode DIGITS causes subsequent numbers to speak out the digits, for example "4 3 2". DOLLARS speaks the numbers as a dollar amount. For example, 432.54 would be spoken as "four hundred thirty-two dollars and fifty-four cents". DAY speaks numbers in the range 1-7 as days of the week; this speak mode is designed to work hand-in-hand with the FoxPro DOW() function. The speak mode specified in an argument list remains active for all subsequent numbers in the argument list until changed. Prompt names given as arguments are in the form of voice file names in the current "AppPrompts" directory, and are in one of several standard digitized audio file formats. The VSpeak() procedure returns codes indicating success or failure, or code indicating success but interrupted by DTMF key press.

In the case of the VSpeak() call on line 48 of the example program in Appendix A, only one argument is provided, specifically the name of a file which contains the prompt "Good-bye, and thank you for calling." In line 51, still another telephony FLL procedure VHangup() is called in order to disconnect the telephony channel.

The pt_call procedure begins on line 62 of the example FoxPro program in Appendix A. In line 71, it calls the VSpeak procedure to speak the prompt in the data file pthello, specifically a hello prompt.

In lines 77-108, the pt_call procedure enters a loop which attempts to obtain a valid 4-digit ID from the caller. The caller is given three tries. In line 78, the procedure calls the telephony FLL function VChkHangup() in order to determine whether the caller has hung up. If so, the procedure returns to line 44. A typical database language program using the telephony
FLL will make frequent calls to VChkHangup() because a caller may hang up at any time.

At line 86, the pt_call procedure calls the VSpeak() telephony FLL procedure to speak the prompt in file ptid, which asks the caller to enter a 4-digit worker ID. In line 87, the pt_call procedure calls another telephony FLL function VGetTones(). The VGetTones() procedure allows the caller to provide information to the program, including menu choices, worker identifications, answers to questions, PIN numbers, and so on. The function takes up to four arguments, beginning with a character variable which upon return will contain the sequence of digits entered by the caller. The second argument is optional and indicates the number of digits to terminate on. The default is 1, and 0 accepts any number of digits. The third argument is optional and indicates a string containing a single character representing a key to terminate on (0-9, *, or #). The default is "space", which indicates no key-specific termination. The fourth argument indicates an inter-digit time-out - the number of seconds to wait between digits before timing out. 0 indicates no inter-digit time-out and the default is 10 seconds. Thus, three different termination mechanisms are available: fixed number of digits, an end-of-input key, and an inter-digit time-out. If more than one mechanism is used, then the first to occur terminates the input. Additionally, VGetTones() will terminate upon a "first-digit time-out" if the caller has not pressed any keys at all within the first five seconds. The database language program can adjust the first-key time-out using a VSetTimeOut() telephony FLL procedure.

On successful termination, VGetTones() returns a value indicating whether termination occurred on inter-
digit time-out, specified key, or number of digits. On failure, VGetTones() returns a value indicating whether termination occurred on first-key time-out, on hang up, or because of bad arguments.

In line 87, the arguments passed to VGetTones are @loc_id, 4, "space" and 15. Thus, it will return after four digits have been entered or after an inter-digit time-out of 15 seconds. There is no key on which it will terminate. The number entered by the caller will be in the FoxPro variable loc_id.

In line 94, the program uses the FoxPro SEEK verb to search the WORKERS database table for the worker ID in loc_id, using the ORDER previously established at line 29. In line 95, the FoxPro function FOUND() is used to determine whether the worker ID entered by the caller exists in the database. If so, then at line 96, the program exits the FOR loop. If not, then at line 104, the program calls VSpeak() with the name of a file ptnoid containing a prompt such as "Sorry, I couldn't find that ID. Please try again." The program allows three attempts for the caller to enter a valid ID; after the third attempt, at line 106, pt_call simply returns.

Once a valid worker ID has been entered, lines 115-183 check each shift individually and for each one, if the worker is scheduled to work, requests confirmation. A count of the number of shifts confirmed is maintained in a FoxPro variable var_num_shifts.

Referring to line 115, pt_call first determines whether the worker is scheduled to work in the early shift on day 1 (es1). If so, then in line 116, pt_call calls another FoxPro procedure query_shift, passing in two parameters: .T. and date1. The first parameter for query_shift indicates whether this is the early or late shift (.T. means TRUE and .F. means FALSE), and the
second parameter is a date string in the form MM/DD/YY. Accordingly, the call in line 116 specifies to query_shift that this is the early shift of the date indicated in datel. The query_shift procedure uses the telephony FILL functions to ask the caller about working the specified shift, and returns true (.T.) or false (.F.) in a variable var_confirm to indicate whether the caller has confirmed that shift. In line 117, the pt_call procedure uses the FoxPro verb REPLACE to update esl with the Boolean value returned in var_confirm. The procedure pt_call performs this operation for each shift of each date in the week.

After all shifts have been queried, the program determines at line 188 whether or not the worker has agreed to work any shifts at all. If not, then in line 189, the program calls VSpeak() with the name of a file ptnoshift containing the prompt, "You are not scheduled to work any shifts in the next week."

Otherwise, at line 198, the program calls VSpeak() with a series of arguments which give the worker an earnings prediction for the coming week. The statement contains five arguments which together assemble the prompt as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ptsigned&quot;</td>
<td>&quot;Thank you. You are signed up to work...&quot;</td>
</tr>
<tr>
<td>var_num_shifts</td>
<td>(The number of shifts confirmed, to be spoken in the standard numeric format)</td>
</tr>
<tr>
<td>&quot;ptshifts&quot;</td>
<td>&quot;...shifts this week, for a total earnings of...&quot;</td>
</tr>
<tr>
<td>&quot;Dollars&quot;</td>
<td>(Subsequent numbers are to be spoken as dollars)</td>
</tr>
</tbody>
</table>
In step 203, pt_call returns to line 44 of the main FoxPro code.

The query_shift procedure used in pt_call begins on line 210 of the example program in Appendix A. In lines 215-217, query_shift first uses the VChkHangup() procedure to exit if the caller has hung up.

In line 224, if this is an early shift, query_shift speaks a prompt by passing arguments to the VSpeak() procedure as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ptearly&quot;</td>
<td>&quot;You are scheduled to work the early shift on...&quot;</td>
</tr>
<tr>
<td>&quot;Day&quot;</td>
<td>(Subsequent numbers are to be spoken as days (Mon., Tues., Wed....))</td>
</tr>
<tr>
<td>DOW(param_date)</td>
<td>Uses the FoxPro DOW() function to convert the specified date to a number 1-7; the number will be spoken as a day.</td>
</tr>
<tr>
<td>DTOC(param_date)</td>
<td>Uses the FoxPro DTOC() function to convert the specified date to a character-type date (e.g. &quot;01/01/94&quot;).</td>
</tr>
</tbody>
</table>

In line 227, if the specified shift is a late shift, then the same prompt is spoken except that it begins with the language in the prompt file ptlea, which is "You are scheduled to work the late shift on...".

In lines 234-268, the caller is given three attempts to enter 1 to confirm, or 2 to reject. Specifically, after checking for hangup (lines 235-237), query_shift calls VSpeak() to speak the prompt "To confirm that you..."
will be available for this shift, press 1. If you will not be available, press 2." In line 242, query_shift calls VGetTones() with arguments indicating termination after one digit or a 15-second inter-digit time-out, with the caller's input to be placed in a variable loc_confirm. In lines 247-251, if the caller pressed the numeral 1 key, then var_confirm is set to .T. and var_num-shifts is incremented. If not, then in lines 253-256, var_confirm is set to .F. If the caller pressed a number that was neither 1 nor 2, then on the first two tries, query_shift calls VSpeak() to speak the prompt, "Please press 1 or 2," (line 263) and the loop repeats. After the third attempt, query_shift sets var_confirm to .F. (line 265) and returns.

It can be seen that the availability of telephony function calls in a database programming language creates a powerful combination which permits interactive voice response system developers to easily allow callers to interact with a database in a very complex manner.

IV. TELEPHONY FLL

The telephony FLL 210 will now be described. For illustration purposes, one of the procedures of the telephony FLL 210 which are callable from the FoxPro language sequencer 206, specifically VSpeak(), will be followed all the way down to the telephony card driver layer 512. A number of additional telephony FLL procedures are described in Appendix B, and their internal operation can be gleaned from Appendices C-G hereto.

A. FoxPro Interface Layer Procedures

Initially, note that FoxPro requires a library extension module to include a structure which defines the entry points and various other characteristics of
each procedure which is callable from the FoxPro language code. In the FoxPro interface layer 502 in
telephony FLL 210-1, this structure is as follows:

```c
FoxInfo myFoxInfo[] = {
    "START", (FPPF) start, CALLONLOAD, ",",
    "STOP", (FPPF) stop, CALLONUNLOAD, ",",
    "VWAITRING", (FPPF) vwaitring, 2, ",1,.I",
    "VHANGUP", (FPPF) vhangup, 0, ",",
    "VCHKHANGUP", (FPPF) vchkhangup, 0, ",",
    "VSPEAK", (FPPF) vspeak, 10,
    "?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.", 
    "VGETTONES", (FPPF) vgettones, 4, ",R,.I,.C,.I",
    "VGETTONESM", (FPPF) vgettonesm, 7,
    "VGETWORDS", (FPPF) vgetwords, 5, ",R,.C,.I,.C,.I",
    "VRECORD", (FPPF) vrecord, 2, ",?.I",
    "VDIAL", (FPPF) vdial, 3, ",C,.I,.R",
    "VDEBUG", (FPPF) vdebug, 1, ",I",
    "VSETDIR", (FPPF) vsedir, 2, ",C,.C",
    "VSENDUSPEAK", (FPPF) vsenddatespeak, 1, ",C",
    "VSENDOUT", (FPPF) vsendtimeout, 2, ",C,.I",
    "VFAXCOVER", (FPPF) vfaxcover, 2, ",C,.?",
    "VFAXDOC", (FPPF) vfaxdoc, 1, ",C",
    "VFAXSENDF", (FPPF) vfaxsend, 2, ",C,.C",
    "VFAXSETUP", (FPPF) vfaxsetup, 2, ",C,.?",
    "VSET", (FPPF) vset, 2, ",C,.?",
};

FoxTable _FoxTable = {
    FoxTable FAR * ) 0,
    sizeof (myFoxInfo) / sizeof (FoxInfo),
    myFoxInfo
};
```

Pertinent parts of the source code for the FoxPro
interface layer are set forth in Appendix C. Pertinent
parts of the VSpeak() procedure will now be described.

VSpeak() begins by obtaining a handle to the current
task and passing it to a function FindSlot(). The
FindSlot() function returns, in a specified pointer
variable sp, a pointer to a DBMS interface layer
telephony line status table line[]. The DBMS interface
layer 502 and the compute layer 504 each maintain such
a table for their own purposes, and the structure of an
entry in the line[] table of the DBMS interface layer is
as follows:

```c
struct line_t {
    boolean isUse;  // is this table slot in use ?
    unsigned int hTask;  // Windows task handle
    int nLineNum;  // line number (not really used)
```
unsigned int handlerid;  // FoxPro event handler ID
HANDLE whandleClientDDE;  // FoxPro handle of hidden client window
HWND hwndClientDDE;       // Windows handle of hidden client window

};

static
struct line_t line[MAX_LINES] = {0};

Each process 204-i maintains a copy of this table, but looks at only the data for itself.

The FindSlot() function merely loops through the above line table until it finds an entry whose hTask field matches the specified task number, and whose bInUse field is true. A pointer to the resulting entry of line[] is returned in sp.

Returning to VSpeak(), a variable play_type is then initialized to play numbers "as numbers". A FOR loop then loops through all of the arguments in the call to VSpeak(). FoxPro passes parameters to extension library procedures using a parameter block consisting of an integer that represents the number of parameter, immediately followed by an array of parameter unions. The first byte of each parameter union indicates that the parameter is passed by reference if it contains the character R, otherwise it is passed by value. If the parameter is passed by value, then the first byte contains a character indicating whether the parameter is a string (C), numeric (N), an integer (I), a date (D), a logical (L), or another type of data. The entire parameter has the following structure:

typedef struct {
  char ev_type;
  char ev_padding;
  short ev_width;
  unsigned short ev_length;
  long ev_long;
  double ev_real;
  MHANDLE ev_handle;
}
VSpeak() begins the parameter loop by determining whether the first parameter is a string. If so, then it copies the length of the string from the ev_length field into a variable nCount, and copies the text of the string into the variable 'text' from the array pointed to by the ev_handle field of the parameter. The procedure then adds a terminating NULL character, trims any trailing blanks, and capitalizes the string.

If the parameter text is one of the predefined numeric format flags "digits", "dollars" or "day", then the VSpeak() procedure sets the flag play_type to indicate how subsequent numeric parameters are to be spoken. It also turns off a play_flag to indicate that the current parameter is not to be spoken.

If the first byte of the current parameter was not a C, but rather was an I, then VSpeak() converts the integer parameter is converted to a string and places it into the variable 'text'. The flag play_flag remains true.

If the first byte of the current parameter is N, then the number contained in the parameter is also converted to text and stored in the variable 'text'.

If play_flag is false with the current parameter, then the loop continues at this point with the next parameter (lines 180, 207). Otherwise, VSpeak() calls a procedure com_Play() either to start a "play list" with the current parameter, or to add the current parameter to an existing play list. The procedure com_Play and related procedures are part of the compute layer 504 and are described in more detail below. Briefly, however, com_Play is called with five arguments: a task handle, which indicates the current Windows task; a call type, which can be either START_PLAY_LIST or ADD_PLAY_LIST or PLAY_PLAY_LIST; a
string which can be either a string of digits or the
name of an audio file; a play_type, which indicates how
numeric parameters are to be spoken; and an
interrupt_mode specifying whether to terminate the
prompt if the caller presses a DTMF key.

After the entire parameter list has been parsed in
the loop, VSpeak() calls com_Play() one more time in
line 213 with the call type PLAY_PLAY_LIST. VSpeak() then prepares a result parameter, which has the same
format as an argument list parameter and returns it to
the FoxPro sequencer 206.

B. Compute Layer Procedures

The compute layer 504 procedure com_Play() will now
be described, as will the compute layer procedures which
are called by com_Play(). Pertinent procedures of the
compute layer source code are set forth in Appendix D.

1. com_Play()
The com_Play() procedure is as follows:

```c
int com_Play(
    const unsigned int hTask,    // task handle (Windows only)
    const int call_type,
    const char *voice_file,      // one filename, a num, or a date
    const int play_type,
    const int interrupt_mode
) {
    int nVRatVal = RC_INTERNAL;  // Voysys return value
    struct line_t *sp;          // ptr to slot in line table
}
```

// find slot in the line table
nVRatVal = FindSlot( hTask, &sp );
if (nVRatVal != RC_SUCCEEDED)
goto done;

// Note: to support speak for online and not online
// should not check for bOnLine here and it will be checked
// in the line driver layer of the server
if (!sp->bConnected)
    { 
    nVRatVal = RC_NOTCONNECTED;
goto done; 
    }

if ( (play_type < AS_NUMBER) || (play_type > AS_DAY) )
    { 
    nVRatVal = -10;
goto done; 
    }
switch (call_type) {
    case IMMEDIATE: // not use for now
        init_play_list(sp);
        if (add_to_play_list(sp, voice_file, play_type) ==
            RC_NOFILE)
            if (add_to_play_list(sp, voice_file, play_type) !=
                RC_SUCCEEDED)
                nRetVal = RC_SPEAKPROMPTFILENAMEFOUND;
        else
            nRetVal = play_play_list(sp, interrupt_mode);
        break;

    case START_PLAY_LIST:
        init_play_list(sp);
        if (add_to_play_list(sp, voice_file, play_type) !=
            RC_SUCCEEDED)
            nRetVal = RC_SPEAKPROMPTFILENAMEFOUND;
        break;

    case ADD_PLAY_LIST:
        sp->nNumItemsInPlaylist++;
        if (add_to_play_list(sp, voice_file, play_type) !=
            RC_SUCCEEDED)
            nRetVal = RC_SPEAKPROMPTFILENAMEFOUND -
            sp->nNumItemsInPlaylist;
        break;

    /*
    * PLAY_PLAY_LIST no longer adds to the play list
    * just plays the list
    */
    case PLAY_PLAY_LIST:
        nRetVal = play_play_list(sp, interrupt_mode);
        break;
    default:
        nRetVal = -12;
    }
}

done:
    return (nRetVal);
}

/* function com_Play */

As can be seen, com_Play() is relatively simple. It merely checks for parameter errors and then calls either
init_play_list(), add_to_play_list() or
play_play_list(), depending on the call type.
Additionally, if the call_type was START_PLAY_LIST, then
after calling init_play_list(), the routine also calls
add_to_play_list().
2. `init_play_list()'

The play list is not actually maintained in the compute layer, but rather it is maintained by the channel server process 214. The entire function `init_play_list()' is merely to assemble an IPC "command" and to pass it to an IPC client layer 506 procedure `ipc_Do_Command()'..

At this point it will be useful to understand that inter-process communications are accomplished in the present embodiment by a client 210-i preparing a command data block and passing it down to the channel server process 214, or by the channel server 214 preparing an event data block and passing it back up to a client process 210-i. The procedure `init_play-list()' uses only the command block, and this structure is defined as follows:

```c
/* format of a command from a higher layer to lower layer */
struct command {
  int  nCommand;    /* command number */
  int  nLine;       /* line number */

  /* syntax inside dialing string is:
   * 0123456789#*ABCD  digit to dial
   * T                use tone dialing (default)
   * P                use pulse dialing
   * W                wait for dial tone
   * ;                pause 2 seconds
   * !_                do a switch hook
   */
  char  string[MAX_DIGITS + 1]; /* dialing string */

  struct play_item; /* item to play, record to,
                     or delete */
  int    path[MAX_DIRS]; /* dirs to search for file
                         (for PLAY) */
  int    dir_num;      /* dir num (for SETDIR,
                        DELETE, RECORD) */

  /* overused field: directory name for setdir, phrase map file
   * name for setfile, syntax map file name for setvocab,
   * syntax name for getwords,
   */
  char  name[MAX_DN + MAX_FN];
```
/* for CALLOUT, max rings to wait before giving up */
/* for WAITFORCALL, rings to wait before answering incoming call */
int answer_rings;  /* for CALLOUT, max rings to wait */

/* for CALLOUT, after answer wait for silence */
boolean wait_for_silence;

/* DTMF digit-string stuff for RECORD, PLAY, GETDTMF */
boolean flush_digits_at_start;  /* flush input at start of operation? */
boolean end_voice_on_any_digit; /* stop voice if get any DTMF digit */
boolean end_voice_on_digit_end; /* stop voice if get end-of-digits */
boolean end_oper_on_any_digit;  /* stop operation if get any DTMF digit? */
boolean end_oper_on_digit_end;  /* stop operation if get end-of-digits? */
boolean end_oper_on_voice_end;  /* stop operation if voice ends? */

/* Values of max_digits:
  * ==0 unlimited (limited only by line card).
  * >0 limited to max_digits digits.
  *
  * Also used to signify max number of words for voice-recognition.
*/
int max_digits;  /* end-of-digits if this many received */
char end_digit;  /* end-of-digits if this digit received */
ulong start_timeout;  /* timeout after voice bef 1st dig (msec) */
ulong total_timeout; /* timeout for whole op (msec); 0 == inf */
ulong interdigit_to; /* timeout after 1st dig between digits (msec) */
boolean get_digits_at_end; /* get input digits at end of operation? */

/* limits for RECORD */
ulong end_silence;  /* this much silence ends recording (msec) */
int total_recording; /* max length of recording (sec) */

/* following are for initializing line in CONNECT command */
boolean do_double_keys; /* detect and report double-keys */
ulong double_key_time; /* double-key max spacing (msec) */
boolean allow_rate_gain; /* allow user to change rate + gain */
int initial_rate; /* initial playback rate */
int initial_gain; /* initial playback gain */
boolean do_pause_compr; /* compress silence to save space */
boolean pause_pure_sil; /* expand comprd pauses to pure silence */
boolean do_AGCl; /* do automatic gain control on input */
/* voice-recognition stuff */
int max_score; /* reject scores > this (lower = better) */
int min_ambiguity; /* reject ambig < this (lower = worse) */
int input_gain; /* input gain (0 - 0x7FFF; def = 0x1000) */

/* set commands */
int io_location; /* input source or output destination - should merge with nSetParm */

/* fax commands */
int nFaxOpCode; /* numeric operation code */
int nFaxParm; /* integer parameter */
char cFaxParm[MAX_FAXPARM+1]; /* string parameter */

/* set/get commands */
int nSetGetOpCode; /* numeric operation code for set and get commands */
ulong dwSetParm; /* unsigned long parameter */
char cSetParm[MAX_SETPARM+1]; /* string parameter */

}

The different commands which can be sent to the lower layer using this command structure are as follows:

/* commands from higher layer to lower layer */
/* all must be consecutive + ascending for debug funcs to work */
#define DVRC_NONE 0 /* no command (placeholder) */
#define DVRC_CALLOUT 7 /* initiate a call and wait for answer */
#define DVRC_CONNECT 1 /* connect Line task to line N */
#define DVRC_DISCONNECT 2 /* disconnect Line task from line N */
#define DVRC_RECORD 3 /* record voice or FAX from user */
#define DVRC_INITPLAY 4 /* initialise playlist to empty */
#define DVRC_ADDDPAY 5 /* add file to playlist */
#define DVRC_PLAY 6 /* play list of voice/FAX files out to user */
#define DVRC_GETDTMF 8 /* get DTMF digit string from user */
#define DVRC_SENDDTMF 9 /* send DTMF digit string out on line */
#define DVRC_ABORT 10 /* abort any operation in progress */
#define DVRC_DELETEFILE 11 /* delete user voice or FAX file */
#define DVRC_HANGUP 12 /* hang up (go on-hook) */
#define DVRC_WAITFORCALL 13 /* wait for incoming call and answer it */
#define DVRC_SETDIR 14 /* set prompt/message directory name */
#define DVRC_GETSTATE 15 /* get information about state of line */
#define DVRC_GETSYSSTAT 16 /* get info about state of entire system */
#define DVRC_SETFILE 17 /* open indexed prompt file */
#define DVRC_SETVOCAB 18 /* open vocabulary map files */
#define DVRC_GETWORDS 19 /* get voice-recognized words from user */
#define DVRC_SETINPUT 20 /* set the sound input source */
#define DVRC_SETOUTPUT 21 /* set the sound output destination */
#define DVRC_FAXCOVER 22 /* set fax cover page */
#define DVRC_FAXDOC 23 /* set fax document pages */
#define DVRC_FAXSEND 24 /* send a fax document */
#define DVRC_FAXSETUP 25 /* setup fax options and parameters */
#define DVRC_SET 26 /* set options and parameters */
#define DVRC_GST 27 /* get options and parameters */

The structure pitem referred to in the definition of
struct command is as follows:

/* format of a "playable item"; it specifies a single prompt or
phrase */

/* tricky: fname[] may contain partial name down to IPC layer,
but is converted to full name in that layer. */

struct pitem {
/* stuff passed down from higher levels */
    int type; /* playable item type */
    char fname[MAX_DN + MAX_FN]; /* voice or FAX file name,
        dir name */
    void far *vdata; /* ptr to voice data in RAM */

/* stuff created and used only at lowest level */
    int ifnum; /* indexed voice file num (-1 = none) */
    int index; /* index into indexed voice file */
};

Given the above, the init_play_list() procedure is
self-explanatory and is as follows:

static
void init_play_list ( struct line_t *sp

40 { struct command cmd;  // command to send
    struct event event;   // event received
    int nVRetVal;        // Voysys return value

    cmd.nCommand = DVRC_INITPLAY;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PI_NONE;
    sp->nNumItemsInPlayList = 0;

    nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
}

3. add_to_play_list()

The compute layer 504 procedure
add_to_play_list() is as follows:
static int add_to_play_list (
    const struct line_t *sp,  // ptr to slot in line table
    const char *voice_file,
    const int play_type
)
{
    int nVRetVal;
    int i, vfile_len;
    char non_digit_char = (char) NULL;

    /* Find the first non-digit, non-blank symbol in the string. */
    /* If there isn't any, speak it as a number. If there is */
    /* one, and it's a slash (/), speak the string as a date. */
    /* Otherwise, treat the string as a file name. */
    vfile_len = strlen (voice_file);
    for (i = 0; i < vfile_len; i++) {
        if ((voice_file[i] < '0') || (voice_file[i] > '9')) &
            // not a digit
            (voice_file[i] == ' ') &
            // space
            (voice_file[i] == '.') &
            // minus sign
            (voice_file[i] == '.') &
            // dot
            non_digit_char = voice_file[i];
        break;
    }

    if (non_digit_char == (char) NULL) {
        /* voice file contains a number value */
        /* Adding number type */
        switch (play_type) {
            case AS_NUMBER:
                nVRetVal = speak_as_number (sp, voice_file);
                break;
            case AS_DIGITS:
                nVRetVal = speak_as_digits (sp, voice_file);
                break;
            case AS_DOLLARS:
                nVRetVal = speak_as_dollars (sp, voice_file);
                break;
            case AS_DAY:
                nVRetVal = speak_as_day (sp, voice_file);
                break;
            default:
                nVRetVal = -2;
                break;
        }
    }
    else if (non_digit_char == '/') {
        nVRetVal = speak_as_date (sp, voice_file);
    }
    else {
        /* voice file contains the name of the voice file to play */
        nVRetVal = addu (sp, voice_file);
    }

    return (nVRetVal);
}
/* function add_to_play_list */
add_to_play_list() can be called with either a number, a date or the name of a voice file to play in the field *voice_file. The field play_type applies only if *voice_file is numeric, so add_to_play_list() first checks to determine whether *voice_file is numeric. If it is, then it calls one of the routines speak_as_number(), speak_as_digits(), speak_as_dollars() or speak_as_day(), depending on play_type. These routines are discussed below.

If *voice_file contains a date, recognized by the presence of the slash (/), add_to_play_list() calls the routine speak_as_date(). Otherwise, *voice_file is assumed to be the name of a voice file to play. The function add_to_play_list() passes this on to yet another function, addu().

The compute layer function addu() is as follows:

```c
static int addu(  
    const struct line_t *sp,  // ptr to slot in line table  
    const char  *voice_file  
)  
{  
    int          nVRetVal;  
    int          path[MAX_DIRS];  
    int          i;  

    path[0] = DIR_APPPROMPTS;  
    path[1] = DIR_UServoICE;  
    path[2] = DIR_TMPFILES;  
    for (i = 3; i < MAX_DIRS; i++)  
        path[i] = DIR_END;  

    nVRetVal = add_voice ( sp, voice_file, path );  
    return ( nVRetVal );  
}  /* function addu */
```

As can be seen, the addu() routine merely creates an array of paths where the specified voice file might be found, and then calls yet another compute layer procedure add_voice(). The routine add_voice() is as follows:

```c
static int add_voice(  
    const struct line_t *sp,  // ptr to slot in line table  
    const char  *voice_file,  
```
const int path[MAX_DIRS]
{
    int nVRetVal;    // Voysys return value
    struct command cmd;
    struct event event;  // event received
    int i;

    if ( strlen ( voice_file ) >= ( MAX_DN + MAX_FN ) ) {
        nVRetVal = RC_BADFILENAME;
        goto done;
    }

    cmd.nCommand = DVRC_ADDPLAY;
    cmd.nLine = sp->nLineNum;
    for ( i = 0; i < MAX_DIRS; i++ )
        cmd.path[i] = path[i];
    cmd.play_item.type = PI_SINGLE_FILE;
    strcpy ( cmd.play_item.fname, voice_file );
    cmd.play_item.index = -1;
    cmd.play_item.vdata = ( void * ) 0xFFFFFFF;

    nVRetVal = ipc_Do_Command ( sp->nTask, cmd, &event );

    done:
    return ( nVRetVal );
} /* function add_voice */

This routine takes as arguments a pointer to the present task's slot in the line table, a pointer to the name of the voice file to be spoken, and an array of paths where the voice file might be found. It sets up a command block using this information and passes it on to ipc_Do_Command().

The routines speak_as_number(), speak_as_digits(), speak_as_dollars(), speak_as_day() and speak_as_date, all essentially dissect the string of characters in *voice_file into individual atomic prompts which are present in voice files in a system prompts directory. These routines all call another compute layer routine addp() with the voice file name for each of the atomic prompts, which merely prepends the path to the system prompts directory and passes the information on to add_voice(). add_voice() sets up the appropriate command block as described above, and passes the individual voice file names on to ipc_Do_Command(). The
compute layer speak_as...() routines and addp() are set forth in Appendix D, together with any utility routines that they call.

4. play_play_list()

The play_play_list() compute layer procedure is as follows:

```c
static int play_play_list (const struct line_t *sp, // ptr to slot in line table
const boolean interrupt_mode // end play if DTMF received?
) {
    struct command cmd; // command to send
    struct event event; // event received
    int nVRetVal = RC_INTERNAL; // Voysys return value

    cmd.nCommand = DVRC_PLAY;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PI_NONE;
    cmd.flush_digits_at_start = TRUE;
    cmd.end_voice_on_any_digit = interrupt_mode;
    cmd.end_voice_on_digit_end = FALSE;
    cmd.end_opc_on_any_digit = FALSE;
    cmd.end_opc_on_digit_end = TRUE;
    cmd.end_opc_on_voice_end = (!interrupt_mode);
    cmd.max_digits = 0;
    cmd.end_digit = '#';
    cmd.get_digits_at_end = FALSE;
    cmd.start_timeout = 0;
    cmd.total_timeout = 0;
    cmd.interdigit_to = 0;
    cmd.end_silence = (unsigned) NOT_APPLIC;
    cmd.total_recording = (unsigned) NOT_APPLIC;

    nVRetVal = ipc_Do_Command (sp->hTask, cmd, &event);

    return (nVRetVal);
}
```

As can be seen, this routine merely sets up a command block with the command DVRC_PLAY and various other parameters, and passes it on to ipc_Do_Command(). ipc_Do_Command() returns the event structure from the channel server 214 in ‘event’, which play_play_list() ignores since it contains no useful information.

C. IPC Client Layer Procedures

Pertinent procedures of the IPC client layer 506 are set forth in Appendix E. The layer includes initialization and termination functions, as well as
procedures to handle commands being passed down from the IPC client layer to the IPC server layer 508, and events being passed up from the IPC server layer 508 to the IPC client layer 506.

As can be seen, the ipc_Do_Command() procedure first initializes an event data structure to values which indicate that no event has been received yet. It then obtains the current client task’s slot in the telephony line table line[] for the IPC client layer 506, and calls the IPC client layer 506 send_message() procedure with the command data structure cmd.

The send_message() procedure merely calls another IPC client layer 506 procedure SendPoke(), which calls the Windows functions to send the command cmd via dynamic data exchange (DDE) in a conventional manner to the DDE window for the present client in server task 214. SendPoke() then awaits a Windows return message using the IPC client layer 506 procedure wait_for_wmessage(). Absent an error, the channel server 214 will always send a return acknowledge message (WM_DDE_ACK).

The wait_for_wmessage() procedure uses conventional Windows functions to periodically poll the Windows event queue until a specified DDE message has been received for the current client task DDE window. It then returns to the calling routines, SendPoke(), send_message(), and, ultimately, ipc_Do_Command().

In the present embodiment, after acknowledging a command, the channel server process 214 always responds to DDE commands with an event structure also sent by DDE. Accordingly, after returning from send_message(), ipc_Do_Command() calls yet another IPC client layer 506 procedure get_message() to wait for and obtain the returned event structure. get_message() is set forth in
Appendix E, and as can be seen, it calls wait_for_wmessage() to await a WM_DDE_DATA message for this task. It then calls another IPC client layer 506 procedure ClientReceiveData(), which sends a WM_DDE_ACK acknowledge message back to the channel server process and places the event structure in the current task's event queue sp->eventq[]. ClientReceiveData() then returns to get_message(), which returns to ipc_Do_Command(), which returns to the compute layer 504 routine which called it.

V. CHANNEL SERVER

As previously described, the channel server 214 is responsible for allocating the telephony and resources of the IVR system in a manner that avoids conflicts. To do this, the IPC server layer and the line driver interface layer of the channel server 214 each maintain a respective static line[] table which contains a large amount of status information about each telephony channel. Each entry in the table corresponds to a different channel. The structure of a single entry in line[] in the IPC server layer 508 is:

```
struct line_t {
    struct command cmd;  // command
    boolean bCmdInProgress;  // is there a command in progress?
    boolean bConnected;  // is there a task attached to line?
    HWND serverwnd;  // server window handle for line
    HWND clientwnd;  // client window handle for line
    boolean bTestMode;  // is the test mode on for this line?

    /* stuff set from dvrc_setdir commands */
    /* dirs voice files are in */
    /* tricky: these names have '\' on the end */
    char dirname[MAX_DIRS][MAX_DN+1];
};
```

The structure of a single entry in line[] in the line driver interface layer 510 is:

```
struct line_t {
    boolean bExists;  /* is this entry used for a line? */
```
int nLineNum; /* line number (0 to MAX_AL-1) */
boolean bConnected; /* is a client connected to line? */
boolean bCmdInProgress; /* is there a cmd in progress? */
int nCmdStage; /* current step of command being executed */

struct command cmd; /* command that is in progress */
struct event event; /* event being constructed */

clock_t timeout; /* if != 0, time to abort cmd */
int nMaxDTMF; /* end play or record if this many digits received */

int nTempVRetVal; /* ultimate value of ep->result */
boolean bSendEvent; /* event filled in; ready to return */

int num_events; /* number of queued events for line */

struct {
    boolean bISWinEvent; /* is this a Windows event? */
    // fields for Windows event
    DWORD dwMsg;
    DWORD dwParam1;
    DWORD dwParam2;
    DWORD dwDevice;
    // fields for Dialogic event
    DWORD nDiaEvent; /* Dialogic event number */
    DWORD nDiaCallState; /* Dialogic call state number */
} queued_ev[MAX_EVENTS_PER_LINE];

boolean bUserIsOffHook; /* is user/caller off-hook? */
boolean bCardIsOffHook; /* card has line off-hook? */
boolean silent; /* for callout, is person silent? */

#define Windows /* Windows */

#endif /* defined(WINDOWS) */

// The following pointer variables should be initialized to all zeros;
long rgb_DosPointer;
short rgb_ProtectedSelector;
long rgb_ProtectedPointer;
long xrgb_DosPointer;
short xrgb_ProtectedSelector;
long xrgb_ProtectedPointer;
long fi_DosPointer;
short fi_ProtectedSelector;
long fi_ProtectedPointer;
long dtmf_DosPointer;
short dtmf_ProtectedSelector;
long dtmf_ProtectedPointer;
long cbp_DosPointer;
short cbp_ProtectedSelector;
long cbp_ProtectedPointer;
long csb_DosPointer;
short csb_ProtectedSelector;
long csb_ProtectedPointer;

int open_file; /* for play/rec, voice file handle */
struct index Files_Index[MAX_PF]; /* for play, for playing playlist */

boolean close_after[MAX_PF]; /* close file after play finishes */
boolean voice_has_ended; /* for play/rec, voice data done? */
boolean digits_have_ended; /* for play/rec, DTMF string done? */

struct ifmap ifile[MAX_IPF]; /* for play, indexed voice files */
int num_openfile; /* for play, num open indexed files */

/* stuff set from initplay, addplay, play commands */
struct pitem plist[MAX_PF]; /* play list items */
int plist_count; /* num files in play list */
};

A. **IPC Server Layer Procedures**

Pertinent routines of the IPC server layer 508 are set forth in Appendix F. The IPC server layer 508 runs a polling loop in the procedure WinMain(), and when a new message has been received, passes it on to either MainWndProc() or DDEWndProc() depending on the specified destination window. When each client starts up, it sends a WM_DDE_INITIATE message by DDE to the server process's main window. This message is received in the IPC server layer's polling loop. In response, the server calls ServerInitiate(), which opens up a hidden window for the particular client and establishes a "conversation" between server and client by placing a correspondence between the two windows in the next available entry of an array Conv[]. Eventually the polling loop will receive a WM_DDE_CONNECT command from the client, with the client requesting telephony line 0. If telephony line 0 is already assigned to a client, the server polling loop will return a "busy" event code. The client will then request connection to line 1, and so on until the first free line is found. When this occurs, the IPC server layer will update the specified entry of its line[] table with an identification of the client and server DDE windows now associated with the channel, and mark the channel as in-use. It then passes the WM_DDE_CONNECT command down to the tm_Do_Command() procedure of the test mode layer 509. Assuming the system is not in test mode, the test mode layer 509 simply passes the command on down to an ld_Do_Command() procedure of the line driver interface layer 510. (It
is assumed herein that test mode is disabled, in which case all calls to the test mode layer are passed directly to the line driver layer. The procedure names are the same, except for a "tm_" prefix for test mode layer procedures and as "ld_" prefix for line driver layer procedures. The two versions of each procedure name are therefore used interchangeably hereinafter and in the appendices.)

If the polling loop receives a WM_DDE_POKE command from a client, then MainWndProc() or DDEWndProc() pass the command on to an IPC server layer routine ServerPoke(), which extracts the command from the message, checks for errors, and calls do_command(). The do_command() procedure begins by setting up a default event structure for return to the client, and then performs various preliminary procedures depending on the command. For the command DVRC_ADDPLAY, do_command() locates the specified voice file in the memory structure of Fig. 1 and assembles a complete path name to the file. It then places the command in the client's entry in the line[] array and sets a "command-in-progress" bit in that entry. The same happens for DVRC_INITPLAY and DVRC_PLAY. do_command() then calls the ld_Do_Command() procedure in the line driver interface layer 510, passing the command structure and an event structure for return information. When ld_Do_Command() returns, do_command() returns the event back to the client using the IPC server layer 508 send_event() procedure. do_command() then unsets the "command-in-progress" bit for the current client, and returns to ServerPoke(). ServerPoke() then sends a WM_DDE_ACK back to the client to acknowledge receipt of the command.

If the polling loop in WinMain() receives a time-out event, then it calls do_p polling(), which calls the line
driver layer to see if any command currently in progress has completed. If so, then the return event structure is transmitted to the client and the command-in-progress flag for that client's entry in the IPC server layer's line[] array is cleared.

B. Line Driver Interface Layer Procedures

The line driver interface layer 510 has four procedures which are callable from the IPC server layer 508: ld_Initialize() (called to start up the layer); ld_Shutdown() (called to shut down the layer); ld_Do_Command() (execute a new command from a client or the next stage of a command already in progress); and ld_Process_Event() (handle an event that came from a telephony card driver using a Windows TAPI service provider). Pertinent ones of these procedures, as well as pertinent support procedures, are set forth in Appendix G.

1. ld_Initialize()

Referring to Appendix G, when the ld_Initialize() procedure is called by the IPC server layer 508, it starts up the Dialogic driver 512, finds out how many telephony channels exist, and initializes them. The Dialogic driver 512 includes a terminate-and-stay-resident (TSR) facility which has been pre-loaded into memory. It is initialized by calling a Dialogic procedure startsys(), which calls a Windows function to generate a software interrupt. The software interrupt vectors to the TSR facility which communicates with the telephony cards 120 via predefined system hardware address ports in a conventional manner. The startsys() function returns the number of telephony channels available. Initialization of the telephony channels
takes place by initializing the line driver interface's array line[] of active line states to initial values.

2. `ld_Process_Event()`

   The routine is pertinent only to certain kinds of telephony card driver software. Events generated by the Dialogic telephony card and driver are passed "up" from the driver layer 512 to the line driver interface layer 510 as described herein. Other kinds of telephony card driver software, however, such as those that use a Windows TAPI service provider, pass events "down" from the top of the server 214 into the IPC server layer 508. These events would be handled by `ld_Process_Event()`. The present embodiment uses the former technique, so `ld_Process_Event()` is never used.

3. `ld_Do_Command()`

   This procedure first uses a routine Check_For_Event() to obtain all events which are waiting in the Dialogic event queue, and transfers them into the server 214's event queue for the appropriate channel numbers. All events pending in the Dialogic event queue are transferred, regardless of the channel number to which they pertain. The server 214's event queues are serviced by the periodic invocation of the IPC server layer 508 do_polling() routine as previously described.

   The `ld_Do_Command()` procedure can be called to start a new command or to perform the next stage of a staged command. The IPC server layer 508 indicates in the argument list to `ld_Do_Command()` whether the present invocation of the procedure is to start or continue a command, the current stage of a staged command being maintained by the line driver layer 510 in its own static line table. In the case of the new command, `ld_Do_Command()` copies the command into the current channel's entry in the line driver layer's static line
table, sets the command stage to 0, and sets a flag for
the entry to indicate that a command is now in progress.
Whether or not this is a new command, the procedure then
'switch'es on the command number.

If the command is DVRC_CONNECT, then a
DoConnectCommand() procedure is called which checks
whether the specified channel is available, updates the
line driver layer's line[] entry for the channel, and
invokes the telephony card driver TSR routines to
establish the connection.

If the command is DVRC_INITPLAY, then the procedure
merely initializes a play list in the line table entry
for the current channel. Since this is a single-stage
command, the routine then turns off the command-in-
progress bit in the entry of the line table, copies an
"EV_INITPLAY" code into an event structure, and returns
to the IPC server layer 508.

If the command is DVRC_ADDPLAY, then the routine
merely adds the provided voice file name to a play list
within the line table entry for the current channel.
Again, it turns off the command-in-progress flag in the
entry, copies the code EV_ADDPLAY into the return event
structure, and returns to the IPC server layer 508.

The DVRC_PLAY command is a multi-stage command. A
number of different stages are set out in Appendix G,
but only stages 0, 11 and 12 need be discussed herein.
Stage 0 indicates the start of a command, and in this
stage, the procedure calls another line driver interface
layer 510 procedure, Play_Playlist(). This is also a
staged procedure, and ld_Do_Command sets the stage to 10
prior to calling Play_Playlist(). Play_Playlist()
changes the command stage for the current channel to
stage 11 as described hereinafter.
The first time Play_Playlist() is called at the start of a command (command stage 10), it loops through all of the items on the play list in the current channel's entry in the static line table, opening each voice file, and determining the DOS file handle, the starting position in the file of the voice prompt, and the length of the voice prompt. Note that the starting position will not always be at the beginning of the file, since the line driver layer 510 supports the presence of multiple indexed prompts in a single file. Play_Playlist() then sets up a table Files_Index[] listing the DOS file handle, the starting position and the length, for each voice prompt in the play list. The format of this table is defined by Dialogic.

Play_Playlist() then builds another Dialogic-defined structure, called a read/write block (My_rwb). Specifically, a pointer to Files_Index[] is written into My_rwb, as are a number of other parameters and pointers, including certain flags which were set by the client in compute layer 504. Such flags include whether speaking should terminate if the caller presses a DTMF key, and if so, which key. Play_Playlist() then passes My_rwb and the current channel number to the Dialogic card TSR driver via the Dialogic xplayf() function. This function will return after the information is transmitted to the telephony cards 120 and will not wait for all of the speaking to complete. In the background, the telephony cards obtain the actual voice data from the voice files via interrupts to the TSR driver as needed. In the meantime, Play_Playlist() sets the command stage in the current channel's entry in the static line table to stage 11 and returns to ld_Do_Command(), which merely returns to the IPC server.
layer 508. The Dialogic TSR will add an event to a Dialogic event queue when speaking has terminated.

As previously described, the IPC server layer 508 do_polling() routine periodically calls ld_Do_Command() for each channel for which a command is in progress. In can be seen in Appendix G that if ld_Do_Command() is called before the Dialogic telephony card has sent an event indicating termination of speaking, then ld_Do_Command() simply returns to the polling routine.

If a Dialogic end-of-playing event has been placed in the event queue for the present channel during the pendency of a DVRC_PLAY command, then the command stage will still be 11. ld_Do_Command() will again call Play_Playlist() which, in stage 11, closes all files that contain only one prompt, and examines the Dialogic event to determine why speaking terminated. Assuming it terminated normally, the procedure sets the command stage to 12 in the current channel's entry in the static line table, sets a flag to indicate whether termination occurred because all prompts were spoken or because of appropriate entry of DTMF digits, and returns to ld_Do_Command(). ld_Do_Command() returns to the IPC server level do_polling() which, because the command is now done, sends an event back to the client to indicate as much.

Accordingly, a telephony voice response system has been described which includes a database language sequencer, a database control module having a plurality of procedures callable by the database language for performing database operations, and a telephony control module having a plurality of procedures callable by the database language sequencer for performing telephony operations. The database operations in the embodiment include reading and writing data to a database. The
telephony operations in the embodiment include speaking a predefined prompt onto a telephony channel, receiving and storing DTMF-encoded input from the telephony channel, and recording audio input from the telephony channel. The database language sequencer calls the database control module procedures and the telephony control module procedures in a sequence defined by a program prepared according to the rules of the database language. The telephony voice response system can control multiple telephony channels by running a separate task for each such channel under a multitasking operating system. A common channel server task is provided which manages the resources of the telephony card for all of the individual channel tasks.

The foregoing description of preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention for various embodiments and with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.
APPENDIX A
Example FoxPro Code
Copyright 1994 Voysys Corporation

1 SET EXACT ON
2 SET NEAR OFF
3 *
4 * Exit from the program if ESC is pressed.
5 *
6 ON ESCAPE RETURN
7 *
8 * Set the library to "voysacce"
9 *
10 SET LIBRARY TO \voysacce\core\voysacce
11 retval = VSet("AppPrompts", \\voysacce\apps\parttime\prompts")
12 retval = VSet("Recordings", \\voysacce\apps\parttime\record")
13 *
14 * Get the hourly rate and the shift length out of the first
15 * record of the
16 * BASEDATA table.
17 *
18 SELECT 1
19 USE BASEDATA
20 GOTO 1
21 var_hourly = laborrate
22 var_hours_per_shift = hrspershift
23 *
24 * Move on to the WORKERS table, which is the one we’ll use for
25 * the phone
26 * conversations.
27 *
28 USE WORKERS
29 SET ORDER TO TAG ID
30 *
31 * To handle calls continuously, go into an infinite loop of
32 * receiving calls.
33 *
34 *
35 DO WHILE .T.
36 * Wait for the phone to ring. Answer it after one ring. Then
37 * call the
38 * pt_call procedure.
39 *
40 retval = VWaitRing(1, 60)
41 DO pt_call
42 *
43 * If the phone has not been hung up, speak the "goodbye" prompt.
44 *
45 IF VChkHangup() = 0
46 retval = VSpeak("ptbye")
47 ENDIF
48 *
49 retval = VHangup()
50 *
51 ENDDO
52 *
53 SET ECHO OFF
54 SET LIBRARY TO
55 RETURN
* pt_call: process one call from a part-time worker.

PROCEDURE pt_call

loc_id = ""
var_num_shifts = 0
var_confirm = .F.

* Say hello.
* 
* retval = VSpeak("phello")
* 
* Loop: ask for a 4-digit worker ID until we get a good one.
* 
* Give the
caller three tries, then hang up.
* 
* FOR var_strikes = 1 TO 3
*   IF VChkHangup() = 1
*     RETURN
*     ENDIF
* 
* Get a 4-digit ID. Set an inter-digit timeout of 15 seconds
* just in
case the caller decides to just sit there.
* 
*   retval = VSpeak("ptid")
*   retval = VGetTones(@loc_id, 4, "", 15)
*   IF retval < 0
*     RETURN
*     ENDIF
* 
* Check the ID in the WORKERS table.
* 
* SEEK loc_id
* IF FOUND()
*   EXIT
*   ENDIF
* 
* If it wasn't found, tell the caller so and try again on the
* first two
* "strikes"; on the third one, hang up.
* 
* IF var_strikes < 3
*   retval = VSpeak("pntoid")
* ELSE
*   RETURN
*   ENDIF
* 
* Go through each of the shifts individually. For each one, if
* the worker is scheduled to work, ask them to confirm. For each
* one that
* they confirm, add 1 to var_num_shifts.
* 
* IF esl
*   DO query_shift WITH .T., datel
*     REPLACE esl WITH var_confirm
* ENDIF
IF ls1
  DO query_shift WITH .F., date1
  REPLACE ls1 WITH var_confirm
ENDIF

IF es2
  DO query_shift WITH .T., date2
  REPLACE es2 WITH var_confirm
ENDIF

IF ls2
  DO query_shift WITH .F., date2
  REPLACE ls2 WITH var_confirm
ENDIF

IF es3
  DO query_shift WITH .T., date3
  REPLACE es3 WITH var_confirm
ENDIF

IF ls3
  DO query_shift WITH .F., date3
  REPLACE ls3 WITH var_confirm
ENDIF

IF es4
  DO query_shift WITH .T., date4
  REPLACE es4 WITH var_confirm
ENDIF

IF ls4
  DO query_shift WITH .F., date4
  REPLACE ls4 WITH var_confirm
ENDIF

IF es5
  DO query_shift WITH .T., date5
  REPLACE es5 WITH var_confirm
ENDIF

IF ls5
  DO query_shift WITH .F., date5
  REPLACE ls5 WITH var_confirm
ENDIF

IF es6
  DO query_shift WITH .T., date6
  REPLACE es6 WITH var_confirm
ENDIF

IF ls6
  DO query_shift WITH .F., date6
  REPLACE ls6 WITH var_confirm
ENDIF

IF es7
  DO query_shift WITH .T., date7
  REPLACE es7 WITH var_confirm
ENDIF

IF ls7
  DO query_shift WITH .F., date7
REPLACE ls7 WITH var_confirm
ENDIF

* Now that we've checked all the shifts, give the output result.
* If there are no shifts, tell the worker so.
* IF var_num_shifts = 0
  retval = VSpeak("ptnoshift")
* On the other hand, if there are shifts scheduled, tell the
* worker so
* and give the earnings prediction. The DOLLARS flag is used to
* speak
* the earnings in dollar format.
* ELSE
  retval = VSpeak("ptsigned", var_num_shifts, "ptshifts", ;
  "Dollars", ;
  var_num_shifts * var_hours_per_shift * var_hourly)
ENDIF
RETURN

*******************************************************************************
* Query_shift: ask the worker about working a particular shift.
* If param_early is TRUE, it is an early shift.
* PROCEDURE query_shift
PARAMETERS param_early, param_date
  loc_confirm = ""*
  IF VchkHangup() = 1
    RETURN
ENDIF

* Tell the caller about the shift. Note that the date must be
* converted to character format before it can be spoken. The DAY
* flag is used to speak the DOW() as a day of the week.
* IF param_early
  retval = VSpeak("ptearly", "Day", DOW(param_date),
  DTOC(param_date))
ELSE
  retval = VSpeak("ptlate", "Day", DOW(param_date),
  DTOC(param_date))
ENDIF

* Now do a loop until we get a 1 (=confirm) or 2 (=won't work)
* from the caller. Give the caller three chances.
* FOR var_strikes = 1 TO 3
  IF VchkHangup() = 1
    RETURN
ENDIF
* Ask for confirmation.
  retval = VSpeak("ptconfirm")
  retval = VGetTones(loc_confirm, 1, " ", 15)
* Check if it's a 1 or a 2. If so, set var_confirm and exit.
* If the shift is confirmed, also increment var_num_shifts.

```
243  * IF loc_confirm = "1"
244     var_confirm = .T.
245     var_num_shifts = var_num_shifts + 1
246     EXIT
247  ENDIF
251  IF loc_confirm = "2"
252     var_confirm = .F.
253     EXIT
254  ENDIF
256  *
257  * If not, remind the caller to press 1 or 2, and
258  * loop again on the first two tries; on the third, assume
259  * non-confirmation.
260  *
261  IF var_strikes < 3
262     retval = VSpeak("pt1or2")
263  ELSE
264     var_confirm = .F.
265     RETURN
266  ENDIF
267  ENDFOR
268
269  RETURN
```
APPENDIX B

TELEPHONY MODULE PROCEDURES

Copyright 1994 Voysys Corporation

2.0 The voysAccess Library - Introduction

The voysAccess library is a set of FoxPro functions created using the FoxPro Library Construction Kit (LCK). The library is called VOYSACCF.FLL. This library must be loaded at the beginning of a voice response program. To do this, the user's program calls:

```
SET LIBRARY TO \...........\voysaccf
```

2.1 The Functions (in alphabetical order)

The following is a list of the voysAccess functions. They are implemented in FoxPro as functions that return an integer return code. All of the functions, and their arguments, are case-independent (VSpeak("HELLO.wav") is the same as vspeak("hello.wav")).

The return codes from the functions are coded as follows:

- Positive integers: Function-specific flavors of success
- Zero: Standard "success" code
- -X or -XX: Function-specific flavors of failure
- -XXX: Standard failure codes

2.1.1 Check for hangup - VChkHangup

Syntax: `retval = VChkHangup()`

Arguments: none

Return Codes: 1 Success, hangup detected
0 Success, no hangup detected

Description: Check whether the caller has hung up on your program. This is a good thing to do at regular intervals. One frequent cause of caller hang-ups is rotary-dial phones, which cannot produce tones and therefore are not useful with voysAccess applications. If the caller has hung up, you should do a VHangup on your side before waiting for the next call.
2.1.2 Dial out - VDial

Syntax:
   retval = VDial( <phone_number> [,<num_rings> ])

Arguments:
   <phone_number>  A character string containing the phone number. The characters
                   "0123456789*#ABCD" will be dialed if included. Special characters allowed in
                   a dialing string are:
                   T  use tone dialing (default)
                   P  use pulse dialing
                   W  wait for dial tone
                   ,  pause for 2 seconds
                   I  do a switch hook
                   All other characters ("-", ",", and ")", for example) in a dialing string will be
                   ignored.
   <num_rings>     The number of times to let the phone ring before deciding to give up
                   (default is 6).

Return Codes:
   0    Success
   -3   Busy
   -4   Rings, but no answer in <=num_rings> rings (RNA)
   -9   Other failure
   -11  Bad "rings" parameter; must be a positive integer

Description: Call out to any phone number. This and VWaitRing are the two ways to start a call. The phone number must be in a character format.
Example: Dial the digit "9" to ask for an outside line, wait two seconds to make sure you get it, then dial Voysys' phone number. Notice that extraneous characters will be ignored.

   retval = VDial("9, (510) 252-1100")

2.1.3 Get Tone (DTMF) Input - VGetTones

Syntax:
   retval = VGetTones( @<variable> [,<number_of_digits>]
                      [,<key_to_terminate_on>]
                      [,<timeout>]
                      [,<use_tone>]
                      [,<use_callback>])
- 58 -


Arguments:
@<variable> A character variable passed by reference. After a successful return, this variable will contain the sequence of digits entered by the caller.

<number_of_digits> Integer; the number of digits to terminate on (0 = any number of digits). Default number of digits is 1.

<key_to_terminate_on> A string containing a single character, which should be a key (0..9, *, or #) to terminate input upon. ("" = no key; this is the default).

<interdigit_timeout> Integer; maximum number of seconds between keys (0 = no interdigit timeout). The default here is 10 seconds.

Return Codes:
2 Success; terminated on inter-digit timeout
1 Success; terminated on specified key
0 Success; terminated on number of digits
-1 Failure; terminated on first-key timeout
-2 Failure; terminated on hangup
-11 Failure; bad termination digits (must be 0-50)
-12 Failure; bad termination key (must be 0-9, *, #, or blank)
-13 Failure; bad interdigit timeout (must be non-negative)

Description: Get one or more tones (DTMF digits) from the caller. This is the caller's primary way of communicating with your application. You must pass in a character variable by reference. The rest of the arguments specify how VGetTones knows when the callers input is done. You have three choices: 1) you may specify a set number of digits (this is good for menu choices and fixed-length data like account numbers), 2) you may specify a key that signals end-of-input (as in "please enter your account number and then press pound"), or 3) you may specify an interdigit timeout (wait N seconds after each digit - if there is a timeout after the first digit, then the input is complete). The last method is a good way of getting variable-length
input out of novice users. Note that in case (2), the key you specify as the termination key does not get included in the string passed back to you by VGetTones. In rarer cases, you may elect to use more than one of these termination methods; put together, they operate on an OR basis - if any of the termination methods is found, input is terminated. The defaults for the termination methods are <number_of_digits> = 1, <key_to_terminate_on> = "", and <interdigit_timeout> = 0; in other words, the complete default is to get one digit and then stop. All examples below assume you have a character variable named "MYVAR". Note that to use the <key_to_terminate_on> argument, you must specify a <number_of_digits> parameter; to use the <interdigit_timeout> argument, you must specify both preceding parameters. VGetTones will return a -1 if the caller has not pressed any keys at all within the first five seconds (5 seconds is a default; this value can be reset with VSetTmOut); this usually indicates a rotary phone or a brain-dead caller.

The timeout issue needs to be carefully considered by the programmer. When prototyping, it is easiest to work without timeouts; this model is closest to PC use, where the system waits patiently for you to finish your input. In voice systems, things are more complicated. One scenario to think about, for example: your program has asked for a 7-digit phone number. If the caller punches in 6 digits, thinking he/she has punched in 7, your program will now wait forever for that seventh digit unless you have a timeout set. For that reason, we have timeouts set (5 seconds for the first key, 10 for inter-digit timeout) on as defaults. This case also shows that you should actively check the return codes from VGetTones. If you had asked specifically for seven digits (as in the second example below), you would get a 0 return code if the caller put in seven digits, or a 2 return code if there was an inter-digit termination timeout. In this case, you want exactly 7 digits, so a return code of 2 should cause you to do the VGetTones again.

Example: Get a single-digit menu choice from the caller (assumes you have a prompt 'menu.wav' recorded that speaks your menu).
- 60 -

`retval = VSpeak("menu.wav")`

`retval = VGetTones(@MYVAR)`

**Example:** Get a seven-digit account number from the caller.

`retval = VGetTones(@MYVAR, 7)`

**Example:** Get a credit card number of arbitrary length, terminated with a # sign, from the caller. Notice that you use 0 (don’t care) for the number of digits.

`retval = VGetTones(@MYVAR, 0, ", ")`

**Example:** Get a credit card number of arbitrary length, terminated when the user pauses for two seconds between digits (two seconds has been found to be a good choice with normal callers; higher values would be appropriate for complete novices). Notice that you use 0 (don’t care) for the number of digits and " " (no terminating key) for the key to terminate on.

`retval = VGetTones(@MYVAR, 0, ", " , 2)`

### 2.1.4 Get Tone (DTMF) Input (Macro) - VGetTonesM

**Syntax:**

```ruby
retval = VGetTonesM(<voice_prompt>, <tries>,
    @<variable>
    [, <number_of_digits>]
    [, <key_to_terminate_on>]
    [, <interdigit_timeout>]
    [, <valid_list>])
```

**Arguments:**

- `<voice_prompt>` A character string containing a file name or memo field name. See the description of the VSpeak function.

- `<tries>` The number of total "tries" to make. In each try, the prompt is spoken, then VGetTonesM waits for input. If it does not get correct input, it tries again, until `<tries>` is exhausted.

- `@<variable>` A character variable passed by reference. After a successful return, this variable will contain the sequence of digits entered by the caller.
<number_of_digits> Integer; the number of digits to terminate on (0 = any number of digits). Default: 1 digit.

<key_to_terminate_on> A string containing a single character, which should be a key (0..9, *, or #) to terminate input upon. (* = no key). Default: no key.

<interdigit_timeout> Integer; maximum number of seconds between keys (0 = no interdigit timeout). Default: 10.

<valid_list> Character; a comma-separated list of legal choices. The default for this argument is no validation list.

Return Codes:
2 Success; terminated on inter-digit timeout
1 Success; terminated on specified key
0 Success; terminated on number of digits
-1 Failure; terminated on initial timeout
-2 Failure; terminated on hangup
-3 Failure; ran out of tries
-11 Failure; bad termination digits (must be 0-50)
-12 Failure; bad termination key (must be 0-9, *, #, or blank)
-13 Failure; bad interdigit timeout (must be non-negative)

Description: VGetTonesM is a "macro" command that combines features of VSpeak and VGetTones. It is meant to address simple demands like "If you're using VISA, press 1. If you're using Mastercard, press 2." and to spare you the necessity of writing loops and input checking. VGetTonesM allows you to specify multiple "tries"; for each try, this function will speak the prompt and then wait for input. If there is no input, or if the input does not match one of the possibilities given in the optional <valid_list>, the function will try again, provided the total number of tries has not been exhausted.

Example: Get a four-digit account number into the variable MYVAR. If the user does not enter any keys, or does not enter enough keys, try one more time before reporting failure. This example assumes that you have a prompt file
"getacct".

reval = VGetTonesM( "getacct", 2, @MYVAR, 4)

Example: Ask the user "For VISA, press 1. For Mastercard, press 2. For Discover, press 3" (assume this is prompt "getcard"), and verify that the input is in fact 1, 2, or 3. Try this three times before giving up.

reval = VGetTonesM("getcard", 3, @MYVAR, 1, " ", 0, "1,2,3")

For more information, please see the descriptions of VSpeak and VGetTones.

Design Notes: Can anyone suggest a better name for this function?

2.1.5 Get Word (Voice Recognition) Input - VGetWords

IMPORTANT NOTE: This function will not be included in the initial developers kit. Also, this design is based on our prototype integration with the VPC board; it may be modified in the future.

Syntax: reval = VGetWords( @<variable> [,<sub-vocabulary_name> [,<number_of_words> [,<word_to_terminate_on> [,<interword_timeout>]]]] )

Arguments:

@<variable> A character variable passed in by reference. After a successful return, this variable will contain the words spoken by the caller. Numbers will be returned in numeric ("1" not "one") form. If there are multiple words, they will not be separated by spaces.

<sub-vocabulary_name> This is the name of the sub-vocabulary set which contains the words to be recognized. The current vocabularies are "yesno" (the words"yes" and "no"), "numbers" (the ten digits, plus "oh" which is interpreted as "zero"), and "all", which has"yes", "no", "oh", and 0-9. If no sub-vocabulary is specified, "all" is used.
- 63 -

<number_of_words> Integer; the number of words to get.
  0 = any number of words.
  The default is one word.

5

<word_to_terminate_on> Character; this is a word to be used as a termination marker (this word is not included in the returned string).
  "" = no word to terminate on; this is the default.

10

<interword_timeout> Integer; the maximum number of seconds between words.
  0 = no timeout; the default is 10 seconds.

20

Return Codes:
  2 Success; terminated on inter-word timeout
  1 Success; terminated on specified word
  0 Success; terminated on number of words
  -1 Failure; terminated on first-word timeout
  -2 Failure; terminated on hangup
  -11 Failure; bad number of words (must be 0-50)
  -13 Failure; bad interword timeout (must be non-negative)

30

-XXX Failure; 50-character word buffer filled up

Description: Get one or more words by voice recognition from the caller. You must pass in a character variable by reference. The second argument lets you choose a sub-vocabulary name from which words will be recognized. The last three arguments specify how VGetWords knows that input is finished. The standard way is to specify a given number of words. You also have the choice, however, of choosing to terminate on a given word, or to terminate if there is a sufficient pause between words.

45

The default first-word timeout is five seconds. In other words, if the caller hasn't said anything within the first five seconds after a VGetWords call, the function will return a -1 (failure) error code.
Example: Get a "yes" or a "no" from the caller. Note that this example uses the default termination condition: get one word.

\[\text{retval} = \text{VGetWords}(@MYVAR, "yesno")\]

Example: Get a 4-digit account number from the caller.

\[\text{retval} = \text{VGetWords}(@MYVAR, "numbers", 4)\]

Example: Get a number of arbitrary length from the caller. Assume the caller has finished reciting the number if there is a 4-second pause after any word. Note that this example uses number of words=0 (any number of words) and terminating word=" " (do not terminate on a particular word).

\[\text{retval} = \text{VGetWords}(@MYVAR, "numbers", 0, " ", 4)\]

Design Notes: In the future, we will want to think about a separator option for this function (having the words un-separated is good for numbers and letters, but bad for real words).

2.1.6 Hang up - VHangup

Syntax: \[\text{retval} = \text{VHangup}()\]

Arguments: none

Return codes: 0 Success

Description: Hang up the line; terminates the call. It is important to remember to do this once a call is complete, to better prepare the system to make or receive the next call.

2.1.7 Record a message - VRecord

Syntax: \[\text{retval} = \text{VRecord}( <filename> | <memo_name> | @<variable> [, <max_recording_length>] )\]

Arguments:

- <filename> Character; the file name for the recording. Unless a full path is specified, this file will be placed
in the recordings directory set with VSetDir (see below).

<memo_name> Character; the name of a memo field where the recording is to be stored. The name of the memo field must be enclosed in quotes.

<variable> Character variable passed by reference. Generate a unique name for the recording and pass it back in the variable. The file will be placed in the directory set with VSetDir.

<max_recording_length> Integer; maximum length of the recording in seconds. Default is 120 seconds.

Return Codes:

0  Success
-1  Failure; timeout
-11 Failure; invalid input (not a character string or character variable passed by reference)
-12 Invalid file name
-XX Invalid recording length; must be a positive integer

Description: Record a message (voice file) from the caller. The file this is to go to can be explicitly specified, or the voysAccess system can generate a unique name.

The second method is generally more appropriate; it is done by passing in a character variable passed by reference. The recording will be terminated after a two-second silence, or if the caller presses ";". Unless otherwise specified, the files will be placed in the RECORDINGS directory set with VSetDir. Recordings can be "embedded" in a database by putting the file name in a character field, or, more directly, by putting the recording into a memo field.

Notice that there is the possibility of a naming conflict between memo fields and file names. For each name passed in quotes that is not obviously a file name, the voysAccess software will first check to see whether it is a valid memo field.
The final parameter to VRecord is the (optional) maximum recording length. The recording will terminate if the length of the message reaches the set maximum. The default is 120 seconds.

Example: Record a message into the file "newmsg.wav"; allow it to be no more than 30 seconds long.

```c
retval = VRecord("newmsg.wav", 30)
```

Example: Record a message into a file name chosen by the system. Get name back in the character variable MYVAR.

```c
MYVAR = " "
MYVAR = VRecord(0)
```

Example: Record a message into the memo field MYMEMO.

```c
retval = VRecord("MYMEMO")
```

2.1.8 Send Tones (DTMF) - VSendTones

Syntax: `retval = VSendTones( <tone_string> )`

Arguments:
- `<tone_string>` A character string containing tones to be sent over the phone line. Valid tone characters are 0..9, *, #, and A..D.

Return codes:
- 0 Success
- XX Invalid tone in tone string

Description: The VSendTones function sends Touch-tone (DTMF) presses over the phone line. It has the same effect as if a human user pressed those keys on his or her phone. This command is mostly useful for creating automated computer-to-computer applications; testing or data transfer, for example.

Example: Play the tones "1", "2", and "3" (analogous to a human user pressing these same keys on their phone):

```c
retval = VSendTones("123")
```
2.1.9 Set voysAccess parameters - VSet

Syntax: retval = VSet( <parameter> , <value> )

Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Allowable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppPrompts</td>
<td>&lt;directory that application prompts are stored in&gt;</td>
</tr>
<tr>
<td>Recordings</td>
<td>&lt;directory that recordings are stored in&gt;</td>
</tr>
<tr>
<td>SysPrompts</td>
<td>&lt;directory that system prompts are stored in&gt;</td>
</tr>
<tr>
<td>DateSpeak</td>
<td>&quot;mmddyy&quot;, &quot;mmdd&quot;, &quot;ddmmyy&quot;, &quot;ddmm&quot;</td>
</tr>
<tr>
<td>VGetTones1stKey</td>
<td>&lt;positive integer&gt;</td>
</tr>
<tr>
<td>TestMode</td>
<td>&quot;on&quot;, &quot;off&quot;</td>
</tr>
</tbody>
</table>

Return Codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-71</td>
<td>Failure; bad parameter (first argument)</td>
</tr>
<tr>
<td>-72</td>
<td>Failure; bad value (second argument)</td>
</tr>
</tbody>
</table>

Description: The VSet function allows the user to set various parameters that affect the behavior of the voysAccess tool kit. All of these settings are also available in the "VOYSACCS.INI" file, with the same names. The rule of precedence is: any use of VSet with a particular parameter overrides the .INI setting of that parameter, but only for the session in which the VSet is used (the .INI file is not changed).

AppPrompts, Recordings, SysPrompts:
These set the directories in which voice files are expected to be found (we use "prompts" to mean voice files that are part of your application, like "for sales, press 1..." and "messages" or "recordings" to indicate voice files recorded by callers). "SysPrompts" sets the directory in which system prompts (like "thirteen" or "April") are to be found. "AppPrompts" sets the directory in which you've placed your application's prompts. "Recordings" sets the directory in which your user messages are recorded. We suggest "\VOYSACCS\APPS\yourapp\PROMPTS" for your application's prompts, and
"\VOYSACCS\APPS\<yourapp>\RECORD" for the recordings. "RECORDINGS" has a direct effect on where the files recorded with VRecord are placed; all three settings are used to set VSPEAKS search order. We suggest that you set your AppPrompts directory (and your Recordings directory, if your application does any recordings) in each of your VOYSACCS applications. You would do this with the following commands:

```c
retval = VSet("AppPrompts",
            "c:\voysaccs\apps\myapp\prompts")
retval = VSet("Recordings",
            "c:\voysaccs\apps\myapp\record")
```

assuming that you followed our suggested directory structure, that your files were on the C: drive, and that the directory name for your application was "myapp". It is generally not necessary to specify where your system prompts are stored; since this is normally fixed, it can be specified in the .INI file.

**Datespeak:**
This sets the way dates are to be spoken by VSPEAK. The default is "mmdyy". The allowable settings are:

```
mmdyy": speak dates as month, day, year
("January 25th, 1994")
mmdd": speak dates as month and day only
("January 25th")
ddmmmyy": speak dates as day, month, year ("the 25th of January 1994")
mmddd": speak dates as day and month only
("the 25th of January")
```

**VGetTones1stKey:**
This sets the first-key timeout for the VGetTones function. This is how long, in seconds, the function will wait if no keys are pressed before declaring a failure. Note that this is different from the interdigit timeout that you can set in VGetTones itself: the inter-digit timeout is how long the function will wait between digit presses.

**TestMode:**
Test Mode allows the programmer to test their program without being connected to a phone line. When TestMode is set to "on", the core telephony functions put up a screen in Windows allowing the user control of their results. For example, VWaitRing function puts up a screen that allows the user to say whether someone called in, or whether there was a timeout.

2.1.10 Speak prompt(s) - VSpeak

Syntax: retval = VSpeak( <value> [,<value>] ... [,<value>] )

Arguments:
VSpeak can take up to 10 <value> parameters.

<value> can be either a value to be spoken, or a code word indicating a "speak mode" for the following argument(s).

Valid values to be spoken:
- any numeric value
- any date (converted to a character string)
- a character string containing a voice file name
- a character string containing the name of a memo field that contains voice data

Valid speak modes:

DOLLARS: speak numeric amounts as dollars and cents
DIGITS: speak numeric amounts as digits
DAY: speak integer amounts as days of the week (1=Sunday)

Return Codes:
1 Success, but interrupted by DTMF keypress
0 Success
-2X Failure: prompt file not found (-21 means first arg, etc.)

Description: VSpeak allows you to speak a sequence of prompts or values. It can take up to 10 arguments. The arguments can be values (numbers, or character strings containing dates or prompt names) or 'speak modes' that regulate how numeric values should be spoken.
Example: To speak the string "We have <number> widgets in our inventory," you would record a voice file containing "We have" (say maybe 'wehave.wav') and another containing "widgets in our inventory" ('widgets.wav'). If WIDGETS was an integer variable or data field containing the number of widgets in inventory, you would then use the FoxPro expression:

```fox
retval = VSpeak("wehave.wav", WIDGETS, "widgets.wav")
```

The "speak modes" only apply to numeric values. The default is a standard numeric format: 432 is spoken as "four hundred thirty two". DIGITS says speak out the digits: 432 would be spoken as "four three two" (this is useful for account numbers and suchlike). DOLLARS speaks numbers as dollar amounts: 432.54 would be spoken as "four hundred thirty-two dollars and fifty-four cents". DAY speaks numbers in the range 1..7 as days of the week; this is designed to work hand-in-hand with the FoxPro DOW( ) function. The speak mode must precede its target(s) in the VSpeak function; putting in a speak mode says "speak all the rest of the numbers in this VSpeak in this format unless I tell you otherwise".

The current version allows you to either specify the file name and the extension ("pname.wav") or just the name ("pname"), assuming that the extension is ".wav". We usually use the latter, the idea being to facilitate portability to other file formats in the future (you could have one version of the file in each format, and tell us once which file format you were using). You may also use the file name plus the dot, but without the extension ("pname.") to tell the voysAccess software that it is a file name.

Notice that there are possible conflicts between voysAccess code words (such as DOLLARS, file names, and memo field names. If a string passed in to VSpeak has letters of the alphabet, but not the dot (.), that would signal a file name, the voysAccess software will first check to see if it's a code word, then if it is a memo, before assuming that it's a file name.
Example: To speak "Today is <day of the week> <date>" you would execute the following FoxPro code (assuming that you have recorded a voice file 'today.wav' with the phrase "Today is..."):  

```
retval = VSpeak("today", "DAY", DOW(DATE()), DTOC(DATE()))
```

Example: To speak the contents of the memo field NAME (if name has voice data inside it):

```
retval = VSpeak("name")
```

2.1.11 Wait for the phone to ring and answer it - VWaitRing

Syntax: retval = VWaitRing( [<num_rings> [, <timeout> ] ] )

Arguments:

- `<num_rings>`: The number of rings to wait before answering. Optional argument; default is 2 rings. Must be a positive integer.
- `<timeout>`: Number of seconds to wait before timing out; 0 means don't time out. Optional argument (you must specify a `<num_rings>` if you want to set a `<timeout>`); default is 0 (no timeout). Must be a non-negative integer.

Return Codes:

- 0: Success (answered a phone call)
- -1: Timeout
- -11: Invalid `<num_rings>` argument
- -12: Invalid `<timeout>` argument

Description: Wait for the phone to ring and answer it. The `<num_rings>` parameter sets the number of rings to wait before answering the phone (1 or 2 are good choices). The `<timeout>` parameter sets the number of seconds to wait before timing out (0 - no timeout - is easiest to work with at the beginning).
APPENDIX C

DB_Fox.c

Copyright 1992-1994 Vovsys Corporation

5

/**************************************************************************
 * 
 * File: DB_Fox.c
 * 
 * Purpose:
 * Set of routines for FLL interface (FoxPro 2.0, 2.5, 2.6)
 * 
 * $Log: /VoysAccess/core/dbi/DB_Fox.C $ 
 * 
 */

#include <dos.h>
#include <errno.h>
#include <math.h>
#include <fcntl.h>
#include <io.h> // _mktemp

25 #include <sys/stat.h>
#include <stdio.h> // Definition of _MAX_PATH
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <dde.h>

30 #include "pro_ext.h"
#define IN_MAIN 1

35 // if define equals to 1, enable fax support
// remember to go the end of this file to uncomment the fax
// functions in FoxInfo structure
// need to define this before the .h files as it could be used in
// the .h header files
#define DO_FAX 0

#include "OS.h"
#include "DVR.h"
#include "Command.h"
#include "Compute.h"

#define MAX_PN 20  // maximum number of characters allowed for program name
#define MAX_LN 127 // maximum number of characters allowed for library name

#define TMPBUFF_SIZE ( 8 * 1024 ) // 8 KB
#define TMP_TEMPLATE "-vXXXXXX"


/*
 * Information about state of each active task.
 * 
 * This data is shared by all client processes, but each process
* looks only at the data for itself.
* Processes find a free slot by looking at the bInUse field.
*/

struct line_t {
    boolean   bInUse;    // is this table slot in use ?
    unsigned int hTask;    // Windows task handle
    int         nLineNum; // line number (not really used)
    unsigned int handlerid;  // FoxPro event handler ID
    WHANDLE    wHandleClientDDE; // FoxPro handle of hidden client
                                 //  window
    HWND       hwndClientDDE; // Windows handle of hidden
                                 //  client window
};

static
struct line_t line[MAX_LINES] = {0};

/****************************************************************************
// Find line table slot that is being used by task hTask
*/

static
int     FindSlot ( const unsigned int hTask, // task handle (Windows only)
                    struct line_t ** psp   // ptr to ptr to slot in line
table )
{
    int nVRetVal;    // Voysys return value
    int nSlot;       // slot in line table

    // find slot in the line table by using hTask
    for ( nSlot = 0 ; nSlot < MAX_LINES ; nSlot++ ) {
        if ( line[nSlot].bInUse &&
             ( line[nSlot].hTask == hTask ) )
            break;
    }

    if ( nSlot >= MAX_LINES ) {
        Debug ( 1, sprintf ( Debug_buf, "DB_Fox/FindSlot: couldn’t
                           find slot in line table" ) );
        nVRetVal = RC_NOTINITIALIZED;    // best guess
goto done;
    }

    *psp = &line[nSlot];

    nVRetVal = RC_SUCCEED;

done:
    return nVRetVal;
}

} /* function FindSlot */
Trim trailing blanks: trim_trailing

This function trims trailing blanks in strings by replacing them
with \'\0\'s. It is needed because FoxPro passes strings filled
with blanks to the end of the field, and because several of our
routines (like "dollars" etc options, plus play of file names) do
not check for trailing blanks.

void trim_trailing(
    char *buf_ptr;
) {char
    for ( buf_ptr = buffer + strlen( buffer ) - 1; *buf_ptr == '
        *buf_ptr = ( char ) NULL;
    } /* function trim_trailing */

GetProgramName

Get filename of currently executing FoxPro program.
Gives base file name, in all caps, without PRG extension.

GetProgramName ( char far *szProgramName ) // result - name of PRG file
{ char
    Value result; // FoxPro result structure
    int nFRetVal; // FoxPro return value
    int nCount;
    nFRetVal = _Evaluate( &result, "PROGRAM()" );
    szProgramName[0] = ( char ) NULL;
    if (nFRetVal == 0) {
        nCount = min ( MAX_PN, result.ev_length );
        _fstrcpy ( szProgramName,
            HandToPtr ( result.ev_handle ),
            nCount );
        szProgramName[nCount] = ( char ) NULL;
        trim_trailing ( szProgramName );
    }
    else {
        Debug ( 1, sprintf ( Debug_buf, "DB_Fox/GetProgramName:
            _Evaluate ret %d", nFRetVal ) );
    }
    Debug ( 3, sprintf ( Debug_buf, "DB_Fox/GetProgramName: ret,
        szProgramName = %s", szProgramName ) );
} /* function GetProgramName */
GetLibraryName
Get filename of currently executing library.

static
void GetLibraryName (char far *szLibraryName, // result - name of LIBRARY file
{
    Value result; // FoxPro result structure
    int nFRetVal; // FoxPro return value
    int nCount;

    nFRetVal = _Evaluate(&result, "SET('LIBRARY')");

    szLibraryName[0] = (char) NULL;
    if (nFRetVal == 0) {
        nCount = min (MAX_LT, result.ev_length);
        _fstrncpy (szLibraryName,
            _HandToPtr (result.ev_handle),
            nCount);
        szLibraryName[nCount] = (char) NULL;
        trim_trailing (szLibraryName);
    }
    else {
        Debug (1, sprintf (Debug_buf, "DB_Fox/GetLibraryName:
            _Evaluate ret %d", nFRetVal));
    }
}

Debug (3, sprintf (Debug_buf, "DB_Fox/GetLibraryName: ret,
    szLibraryName 's', szLibraryName));

FoxPro event handler: BvtHandler

int FAR BvtHandler (WHAT theWindow, ..EventRec FAR *theEvent
{
    struct event event; // event to send down
    UINT message; // component of Windows message
    WPARAM wParam; // component of Windows message
    LPARAM lParam; // component of Windows message
    struct line_t *sp; // ptr to slot in line table
    int nVRetVal; // Voysys return value

    // find slot in the line table
    nVRetVal = FindSlot (GetCurrentTask (), &sp);
if ( nRetVal != RC_SUCCEEDED )
    return NO; // FoxPro should handle the event

if ( theWindow != sp->whandleClientDDE )
    return NO; // FoxPro should handle the event

// extract fields from FoxPro form to Windows form
message = ( UINT ) ( theEvent->message );
wParam = ( WPARAM ) ( theEvent->misc );
lParam = ( LPARAM ) ( theEvent->misc2 );

switch ( message ) {
    case WM_DDE_ACK: case WM_DDE_DATA: case WM_DDE_TERMINATE:
        // we will handle the event
        break;
    default:
        // event we don't care about
        return YES; // FoxPro should NOT handle the event
}

event.nEvent = EV_DDE;
event.nLine = 0;
event.C_error = 0;
event.DV_error = 0;
event.DOS_error = 0;
event.string[0] = ( char ) NULL;
event.filename[0] = ( char ) NULL;

// pack fields from Windows form to voysAccess form
event.dde[0] = message;
event.dde[1] = wParam;
event.dde[2] = lParam;

nRetVal = com_ProcessEvent ( sp->hTask, event );
if ( nRetVal < RC_SUCCEEDED ) {
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/EvtHandler: event handling failed\n" ) );
}

return YES; // FoxPro should NOT handle the event

/* function EvtHandler */

/* Put memo field data into a temporary file and return the filename if success. */

int IsMemoField {
    char *field,
    boolean *bMemoField,
    Value *val,
    Locator *loc

    int nRetVal;
}
NTI nNameTblIndex;
    *bMemoField = FALSE;

      // try to find out if this is a memo field
      nNameTblIndex = _NameTableIndex ( (char far *) field );
      if ( nNameTblIndex != -1 ) {
          // name is in the name table
          if ( _FindVar( nNameTblIndex,
                         0,      // database area number
                         (Locator FAR *) loc
                     ) ) {  // variable with the name exist
              if ( _Load ( (Locator FAR *) loc, (Value FAR *)
                         val ) != 0 ) {
                  nRetVal = RC_FOXPRESSOR;
                  goto done;
              }
              if ( val->ev_type == 'M' ) { // memo field
                  // this is a memo field
                  if ( val->ev_width <= 0 ) {
                      nRetVal = RC_FOXPRESSOR;
                      goto done;
                  }
              }
              *bMemoField = TRUE;
          }
          nRetVal = RC_SUCCEED;
      }

done:
    return ( nRetVal );

} /* function IsMemoField */

/**************************************************************************/

Get temporary filename.

/**************************************************************************/

int GetTempFilename (  
    const unsigned int hTask,  // task handle (Windows only)  
    const int nFileType,  
    char *pTmpFile  
 )
{
    int nRetVal;
    ulong dwDummy;
    char szTmpName[MAX_FN];  // temporary filename

    // Assumption: pTmpFile has the size of MAX_DN + MAX_FN

    // Attempt to find a unique filename
    strcpy ( szTmpName, TMP_TEMPLATE );
    if ( _mktemp ( szTmpName ) == NULL ) {
        nRetVal = RC_OSERROR;
        goto done;
    }
} // get the temporary file path from server
nVRetVal = com_Get ( hTask, nFileType, ( char far * ) pTmpFile, &dwDummy );
if ( nVRetVal != RC_SUCCEED )
goto done;
strcat ( pTmpFile, szTmpName );
strcat ( pTmpFile, "*.TMP" );
nVRetVal = RC_SUCCEED;
done:
return ( nVRetVal );
} /* function GetTmpFilename */

/******************************************************************************
 Put memo field data into a temporary file.
******************************************************************************/
int PutMemoDataInTmpFile ( const unsigned int hTask, // task handle (Windows only)
 const char *filename,
 const Value val
)
{
char szTmpFile[MAX DN + MAX_FN];
int nVRetVal;
FCHAN MHandle, PHandle;
long lStartPos;
long lMemoSize;
char *pBuff;
int nReadSize, nWriteSize;

MHandle = ( FCHAN ) val.ev_width;

// position the file pointer on the beginning of the section
// that belongs to the current record
lStartPos = ( long ) val.ev_real;
_FSeek ( MHandle, lStartPos, FS_FROMBOF );

mh = _AllocHand ( TMPBUFF_SIZE );
if ( mh == 0 )
{
nVRetVal = RC_RAMFULL;
goto done;
}
pBuff = ( char * ) _HandToPtr ( mh );
nRetVal = GetTmpFilename ( hTask, GET_TMPFILES, ( char * ) &szTmpFile );
if ( nRetVal != RC_SUCCESS )
goto finish2;

// open the temporary file
FHandle = _FCreate ( ( char far * ) &szTmpFile, FC_NORMAL );
if ( FHandle == -1 ) {
    nRetVal = RC_FILEERROR;
goto finish2;
}

while ( lMemoSize > 0 ) {
    nReadSize = (int) min ( (long) TMPBUFF_SIZE, lMemoSize );
    nReadSize = _FRead ( MHandle, ( char FAR * ) pBuff,
                        nReadSize );
    lMemoSize -= nReadSize;

    nWriteSize = _fwrite ( FHandle, ( char FAR * ) pBuff,
                          nReadSize );
    if ( nReadSize != nWriteSize ) {
        nRetVal = RC_DISKFULL;
goto finish1;
    }
}

strcpy ( filename, ( char * ) &szTmpFile );
nRetVal = RC_SUCCESS;

finish1:
    _FClose ( FHandle );

finish2:
    _FreeHand ( mh );
done:
    return ( nRetVal );
} /* function PutMemoDataInTmpFile */

/**************************************************************************
* Get memo field data from a temporary file.  */
***************************************************************************/

int GetMemoDataFromTmpFile ( char *filename,
                             const Value val,
                             const Locator loc )
{
    int nRetVal;  // Vosys return value
    long lFileSize;
    FCHAN MHandle, FHandle;
    long lStartPos;

    // open the temporary file
    FHandle = _FOpen ( ( char far * ) filename, FO_RDONLY );

if ( FHandle == -1 ) {
    nVRetVal = RC_FXPROERROR;
    goto done;
}

// find the size of the file
lFileSize = _FSeek ( FHandle, 0L, FS_FROMEOF );

// get the file channel for memo field
MHandle = ( FCHAN ) Val.ev_width;

if ( ( lStartPos = _AllocMemo ( ( Locator FAR * ) &loc, 
    lFileSize ) ) == -1 ) {
    nVRetVal = RC_FXPROERROR;
    goto finish;
}

// position the file pointer on the beginning of the section
// that belongs to the current record
_FSeek ( MHandle, lStartPos, FS_FROMEOF );

// do the copy from temporary file to memo field
nVRetVal = _FCopy(MHandle, 1StartPos, FHandle, 0L, lFileSize);
if ( nVRetVal == 0 ) {
    nVRetVal = RC_FXPROERROR;
    goto finish;
}

nVRetVal = RC_SUCCESS;

finish:
_FClen ( FHandle );

done:
return ( nVRetVal );
} /* function GetMemoDataFromTmpFile */

/*******************************************************************************
Startup the system: START
This routine is called automatically when the library is loaded.
It does the "init" and "connect" calls.
&& We may want to leave "connect" to the user.
*******************************************************************************/

void FAR start ( )
{
int nVRetVal; // Voysys return value
struct line_t *sp; // ptr to slot in line table
int nSlot; // slot in line table
char szProgramName[MAX_PN + 1];
ulong dwFileVersion;
char szDir[MAX_DN];

// send a beep to the PC speaker to indicate this "start"
//function has been called
MessageBeep ( (UINT)-1 );
#if !NO_DEBUGGING_AT_ALL
#define DEBUG INTO FILE
#if DEBUG INTO FILE
/* delete, create and open debug file */
{
    int i, j;
    char name[13];
    int filehd1;

    i = 0;
    j = 0;

    // debug file name: vcdebug00.txt, vcdebug01.txt, ..., \n    // vcdebug98.txt, vcdebug99.txt
    // 012345678901
    strcpy ( name, "vcdebug00.txt" );
    while ( TRUE ) { if ( ( i > 9 ) ) {
        // check to see there is a need to increment the
        // 1st digit of the filename
        i = 0;  // reset the 2nd digit to 0
        j++;   // increment 1st digit
        name[6] = '0' + j;
    }
    name[7] = '0' + i;

    // if exists, use another name
    filehd1 = open ( name,
        O_TEXT | O_RDONLY,   // |O_EXCL,
        S_IRREAD
    );

    if ( ( filehd1 != -1 ) ) {
        // open success, the file exist and can be openned
        // so we should try the next sequence number
        close ( filehd1 );

        if ( ( ( i != 9 ) || ( j != 9 ) ) ) { // not reach the last file "vcdebug99.txt" in
            // the sequence yet
            // so go back and try the next sequence number
            i++;   // increment the 2nd digit of the filename
            continue;
        } else { // reach the last file "vcdebug99.txt"
            // so start from "vcdebug00.txt" again
            name[6] = '0';
            name[7] = '0';
            i = 0;
            j = 0;
        }
    }

    // open fail, the file does not exist OR
    // file exist but we already reach the last file
    // "vcdebug99.txt"
    // so we try to create/truncate the file and start a
    // new debug file
    debugfile = open ( name, O_CREAT | O_TRUNC | O_TEXT |
if (debugfile != -1)
    break;  // success, file is created
if (errno != EACCES) {
    // failure other than "busy"
    nRetVal = RC_OSError;
    goto done;
}

// failure is "busy" so that means the file is being
// used right now
// go back to try the next sequence number
i++;  // increment the 2nd digit of the filename
}
#endif

Debug_flag = 3;
#endif

GetProgramName(szProgramName);

// find slot in the line table
nRetVal = FindSlot(GetCurrentTask(), &sp);
if (nRetVal == RC_SUCCESS) {
    Debug(1, sprintf(Debug_buf, "DB_Fox/start: task has already started.");
    nRetVal = RC_ALREADYINITIALIZED;
    goto done;
}

// find a free slot in the line table
for (nSlot = 0; nSlot < MAX_LINES; nSlot++) {
    if (!line[nSlot].bInUse)
        break;
}
if (nSlot == MAX_LINES) {
    Debug(5, sprintf(Debug_buf, "DB_Fox/start: line table is full.");
    nRetVal = RC_SWLIMIT;
    goto done;
}

sp = &line[nSlot];
sp->bInUse = TRUE;
sp->hTask = GetCurrentTask();
sp->nLineNum = MAX_LINES;
sp->handlerId = ActivateHandler(RvtHandler);
if (sp->handlerId == 0) {
    nRetVal = RC_OSError;
    sp->bInUse = FALSE;
    goto done;
}

sp->hwndClientDDE = _WOpenP(10, 10, // top, left
                           20, 20,  // bottom, right
                           WEVENT,  // gets events?
                           WINDOW_SCHEMA, // default color scheme
                           (Scheme FAR *) 0, // ignored
if ( sp->whandleClientDDE == 0 ) {
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: _WOpenP
failed" ) );
    nVRetVal = RC_OSError;
    _DeActivateHandler ( sp->handlerid );
    sp->handlerid = 0;
    sp->bInUse = FALSE;
    goto done;
}

sp->hwndClientDDE = hWndToHwnd ( sp->whandleClientDDE );
if ( !sp->hwndClientDDE ) {
    nVRetVal = RC_OSError;
    _DeActivateHandler ( sp->handlerid );
    sp->handlerid = 0;
    _WClose ( sp->whandleClientDDE );
    sp->whandleClientDDE = 0;
    sp->bInUse = FALSE;
    goto done;
}

/*
 Call the init routine.
*/

nVRetVal = com_Initiate ( sp->hTask, 0, sp->hwndClientDDE );
if ( nVRetVal != RC_SUCCEEDED ) {
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: init
routine failed" ) );
    _DeActivateHandler ( sp->handlerid );
    sp->handlerid = 0;
    _WClose ( sp->whandleClientDDE );
    sp->whandleClientDDE = 0;
    sp->bInUse = FALSE;
    goto done;
}

/*
 Call the "connect" routine.
*/

nVRetVal = com_Connect ( 
    sp->hTask, 
    MAX_LINES, /* request any line */
    &(sp->nLineNum) /* retval -- line we got */
);
Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot get FILEFORMATS* ) );
strcpy ( szDir, DEF_DIR_SYSPROMPTS );
if ( dwFileFormat & FM_VOX )
strcat ( szDir, FM_VOX_STR );
else
  if ( dwFileFormat & FM_WAV )
    strcat ( szDir, FM_WAV_STR );

/*
 * Initialize the default voice directories.
 * Ignore any failures.
 */
nRetVal = com_Set_Directory ( sp->hTask, "SYSPROMPTS", szDir );
if ( nRetVal != RC_SUCCESS )
  Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot set SYSPROMPTS to DEF_DIR_SYSPROMPTS* ) );
nRetVal = com_Set_Directory ( sp->hTask, "APPROMPTS", DEF_DIR_APPROMPTS );
if ( nRetVal != RC_SUCCESS )
  Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot set APPROMPTS to DEF_DIR_APPROMPTS* ) );
nRetVal = com_Set_Directory ( sp->hTask, "RECORDINGS", ".*" );
if ( nRetVal != RC_SUCCESS )
  Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot set RECORDINGS to ..*" ) );
nRetVal = com_Set_Directory ( sp->hTask, "FAXFILES", ".*" );
if ( nRetVal != RC_SUCCESS )
  Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot set FAXFILES to ..*" ) );
nRetVal = com_Set_Directory ( sp->hTask, "TEMPFILES", ".*" );
if ( nRetVal != RC_SUCCESS )
  Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot set TEMPFILES to ..*" ) );
nRetVal = com_Set_Vocabulary ( sp->hTask, DEF_DIR_VOCABS );
if ( nRetVal != RC_SUCCESS )
  Debug ( 1, sprintf ( Debug_buf, "DB_Fox/start: cannot set Vocabulary to DEF_DIR_VOCABS*" ) );

/*
 * Set the voice recognition vocabulary.
 * Ignore any failure.
 */
// nRetVal = com_Set_Vocabulary ( sp->hTask, "demo" );
nRetVal = RC_SUCCESS;

// send a beep to the PC speaker to indicate this "start"
// function has been initialized
MessageBeep ( (UINT)-1 );
done:
// No return code; return back to FoxPro.
return;
}

) /* function start */

/*****************************/

Shut down the system: STOP
This routine is called automatically when the library is unloaded or a QUIT is done out of FoxPro. It does the "disconnect" and "shutdown" calls. We may want to leave "disconnect" to the user.

```
void FAR stop ( )
{
    struct line_t *sp;   // ptr to slot in line table
    int nRetVal;         // Voysys return value

    // find slot in the line table
    nRetVal = FindSlot( GetCurrentTask ( ), &sp );

    if ( nRetVal != RC_SUCCEED )
    {
        goto done;
    }

    com_Disconnect ( sp->hTask );
    com_Shutdown ( sp->hTask );

    if ( sp->handlerid != 0 )
    {
        DeActivateHandler ( sp->handlerid );
        sp->handlerid = 0;
    }

    if ( sp->chandleClientDDE != 0 )
    {
        WClose ( sp->chandleClientDDE );
        sp->chandleClientDDE = 0;
    }

    sp->nLineNum = MAX_LINES;
    sp->bInUse = FALSE;

    done:

    #if !NO_DEBUGGING_AT_ALL
    #if DEBUG INTO FILE
        close ( debugfile );
    #endif
    #endif

    // No return code; return back to FoxPro.
    // no debug stmt; debug file is closed return;
}
```

---

Wait for Ring: VWAITRING

```
recode = VWAITRING( [<num_rings>(I) [,<timeout>(I)>] )
```

Returns:

- 0 = Success
- 1 = Timeout
-11 = Invalid number of rings (must be positive integer)
-12 = Invalid timeout (must be non-negative integer)
-10X = Standard error codes (from dVR.h)

***
void FAR vwaitring (  
    ParamBlk FAR *parm  
    )  
{  
    int num_rings;  // number of rings before answer  
    int timeout;  // timeout value in seconds  
    int nVRetVal;  // Voysys return Value  
    struct line_t *sp;  // ptr to slot in line table  
    // find slot in the line table  
    nVRetVal = FindSlot( GetCurrentTask ( ), &sp );  
    if ( nVRetVal != RC_SUCCEED )  
        goto done;  
    /  
    /* Extract the arguments to the function. Remember that both  
    * arguments are optional. &&& This does not yet take  
    * into account the default settings.  
    */  
    if ( parm->pCount >= 1 )  
        num_rings = ( ( int ) parm->p[0].val.ev_long );  
    else  
        num_rings = DEF_WAITRING_NUM_RINGS;  
    if ( parm->pCount >= 2 )  
        timeout = ( (int) parm->p[1].val.ev_long );  
    else  
        timeout = DEF_WAITRING_TIMEOUT;  
    /*  
    * Call the *answer* function.  
    */  
    nVRetVal = com_Answer ( sp->hTask, num_rings, timeout );  
    if ( nVRetVal == RC_TOTALTIMEOUT )  
        nVRetVal = RC_WAITRINGETIMEOUT;  
    done:  
    // Send the return code back to FoxPro.  
    MemFill ( &result, 0, sizeof ( result ) );  
    result.ev_type = 'I';  
    result.ev_long = nVRetVal;  
    RetVal ( &result );  
    return;  
}  /* function vwaitring */

}/*----------------------------------------------------------------------
Hang up:  VHANGUP  
recode = VHANGUP()  

Returns:  
0 = Success  
-10X = Standard error codes (from dVR.h)  
}*/
void FAR vhangup ( )  
{
int nVRetVal;  // Voysys return value
Value result;   // FoxPro return structure
struct line_t *sp;  // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
if ( nVRetVal != RC_SUCCEEDED )
goto done;

/*
 * Call the hangup routine.
 */
NvRetVal = com_Hangup ( sp->hTask );

done:
// Send the return code back to FoxPro.
_MemFill ( &result, 0, sizeof ( result ) );
result.ev_type = 'I';
result.ev_long = nVRetVal;
_ReturnVal ( &result );
return;

} /* function vhangup */

.TODO

**************************************************************************
Check for hangup:  VCHKHANGUP

retcode = VCHKHANGUP()

Returns:
1    = Success, hangup detected
0    = Success, no hangup detected
-10X = Standard error codes (from DVR.h)

**************************************************************************

void FAR vchkhangup ( )
{
int nVRetVal;  // Voysys return value
Value result;   // FoxPro return structure
struct line_t *sp;  // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
if ( nVRetVal != RC_SUCCEEDED )
goto done;

/*
 * Call the check for hangup routine.
 */
NvRetVal = com_Hangup_Detect ( sp->hTask );

done:
// Send the return code back to FoxPro.
_MemFill ( &result, 0, sizeof ( result ) );
result.ev_type = 'I';
result.ev_long = nVRetVal;
_ReturnVal ( &result );
return;
void FAR vspeak ( 
    ParamBlk FAR *parm
)
{
    int nRetVal;            // Voysys return value
    Value result;          // use to check memo field and
    FoxPro return structure <<<
    int play_type;         // Play type (DIGITS, DOLLARS, etc
    int interrupt_mode;    // 0=no interrupt, 1=interrupt OK
    char text[MAX_DN + MAX_FN];
    char text[MAX_FN + 1];
    int arg_count;
    int play_flag;         // Should this argument be played?
    int list_started;      // have we started a play list yet?
    Double dval;
    long lval1, lval2;
    struct line_t *sp;     // ptr to slot in line table
    int nCount;
    boolean bMemoField = FALSE;
    Locator loc;
    int nNumMemoField = 0;
    int nPlaylistLen = 1;
    char *pFileName;
    MHANDLE mh[MAX_PLAYLISTLEN];
    char FAR *lpName;

    // find slot in the line table
    nRetVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nRetVal != RC_SUCCESS )
        goto done;

    /*
    * Initialize the play type to AS_NUMBER (the default).
    * /
    play_type = AS_NUMBER;

    /*
    * Set the interruptibility (default is ON) &&& Need to check
    * a table here
    */
interrupt_mode = 1;

/*
 * Set list_started to FALSE; we have not yet initialized a
 * play list.
 * Set the return code to 0 (success).
 */
list_started = FALSE;

/*
 * Process the arguments one at a time.
 */
for ( arg_count = 0; arg_count < parm->pCount; arg_count++ ) {
    if ( nPlayListLen > MAX_PLISTLENGTH ) {
        nRetVal = RC_PLISTLENGTH;
        goto done;
    }

    pFileName = text;

    /*
     * Set the "play flag" for this argument to TRUE.
     */
    play_flag = TRUE;

    /*
     * For each argument, check the type. If it is a string,
     * we need to extract it, which involves conversion to
     * a far pointer followed by a copy to our "text"
     * buffer.
     */
    if ( parm->p[arg_count].val.ev_type == 'C' ) {
        nCount = min(MAX_FN,parm->p[arg_count].val.ev_length);
        _fstrncpy ( ( char far * ) text,
            _HandToPtr ( parm->p[arg_count].val.ev_handle ),
            nCount );
        text[nCount] = ( char ) NULL;
        trim_trailing ( text );

        /*
         * Now capitalize the string.
         */
        _strupr ( text );

        /*
         * Check if the string is really one of our modifiers
         * (DOLLARS, DIGITS, DAY). If it is, set the
         * "play_type" to the correct value, and turn off
         * the "play flag".
         */
        if ( !strcmp ( text, "DIGITS" ) ) {
            play_flag = FALSE;
            play_type = AS_DIGITS;
        }
        else if ( !strcmp ( text, "DOLLARS" ) ) {
            play_flag = FALSE;
            play_type = AS_DOLLARS;
        }
        else if ( !strcmp ( text, "DAY" ) ) {
            play_flag = FALSE;
            play_type = AS_DAY;
        }
    }
}

done:
else {
  // try to find out if this is a memo field
  nVRVal = IsMemoField (text, &bMemoField, &result, &loc);

  if ( nVRVal != RC_SUCCEED )
    goto done;

  if ( bMemoField ) {
    // allocate memory for filename
    mh[nNumMemoField] = _AllocHand ( MAX_DN + MAX_FN );
    if ( mh[nNumMemoField] == 0 ) {
      nVRVal = RC_RAMFULL;
      goto done;
    }

    lpName = _HandToPtr ( mh[nNumMemoField] );

    nVRVal = PutMemoDataInTmpFile ( sp->hTask,
                                   (char *) lpName, result );
    if ( nVRVal != RC_SUCCEED )
      goto done;

    pFileName = (char *) _fstrchr ( lpName, '\\ '); // get the
                // filename only
    pFileName++;
    // skip the 'e' char

    nNumMemoField++;
  }
}

} else if ( parm->p[arg_count].val.ev_type == 'I' ) {
  /*
   * If the argument is of some other type, convert it
   * to text. && Always using two decimal places
   * for type N. &&
   */
  sprintf (text, "\%d", parm->p[arg_count].val.ev_long);
}

} else if ( parm->p[arg_count].val.ev_type == 'N' ) {
  /*
   * The Fox LCK has a bug that freezes if "sprintf" is
   * used with a "%f". This code is a work-around.
   */
  dvale = parm->p[arg_count].val.ev_real;
  lval1 = (long) floor ( dvale );
  lval2 = (long) (100. * (dvale - floor (dvale)) + .5);

  sprintf (text, "%ld.%ld", lval1, lval2);
}

/*
 * If the play flag is on, play this argument. If it is
 * the first one (list_started = FALSE), then we need
 * to start a play list.
 */
if ( play_flag ) {
  if ( !list_started ) {
    nVRVal = com_Play (sp->hTask,
START_PLAY_LIST,
pFileName,
play_type,
interrupt_mode
);

if ( nRetVal != RC_SUCCEED )
goto done;
list_started = TRUE;
}
else {
    nRetVal = com_Play(
        sp->hTask,
        ADD_PLAY_LIST,
pFileName,
        play_type,
        interrupt_mode
    );

    if ( nRetVal != RC_SUCCEED )
goto done;
}

nPlayListLen++;
}
} /* for loop - go to next argument */

/*
 * Now play the whole list, as long as the return code is
 * still zero.
 */

nRetVal = com_Play(
    sp->hTask,
    PLAY_PLAY_LIST,
    "NONE",
    play_type,
    interrupt_mode
);

done:
if ( nNumMemoField > 0 ) {
    // delete the temporary field(s)
    for ( nCount = 0; nCount < nNumMemoField; nCount++ ) {
        if ( remove ( ( char * ) _HandToPtr ( mh[nCount] ) ) <
            0 ) {
            // may not need to return error code
            Debug ( 1, sprintf ( Debug_buf,
                "DB_Fox/vspeak: cannot remove temporary file '\"",
                ( char * ) _HandToPtr ( mh[nCount] ) ) );
            // just continue
            
            _FreeHand( mh[nCount] );
        }
    }
    // Send the return code back to FoxPro.
    MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = nRetVal;
    _RetVal (&result);
    Return;
}
} /* function vspeak */
/***************************/
Get touch tones (DTMF): VGETTONES

retcode = VGETTONES( @<variable>(C) [,<number_of_digits>(I)]
[,<key_to_terminate_on>(C) ]
[,<interdigit_timeout>(I) ] )

The first (mandatory) parameter is a character variable to put
the received keys (0-9, #, *) into. The next three parameters
(optional) set the termination conditions. This can be one of
more of:
- a set number of digits (0=any number of digits)
- a key to terminate on (like #) which is not to be included
  in the output (* = no key)
- an interdigit timeout (0=no interdigit timeout)

The absolute defaults are 1 keystroke, no terminating key, 10
second interdigit timeout. These defaults can be overridden by
the use of another function. THOSE defaults, in turn, are
overridden by any use of the optional arguments to this function.

Returns:
2 = Terminated on interdigit timeout (success)
1 = Terminated on specified key (success)
0 = Terminated on number of digits (success)
-1 = Terminated on timeout (failure)
 2 = Terminated on hangup (failure)
-11 = Bad termination digits (must be 0-MAX_DIGITS)
-12 = Bad termination key (must be 0-9, #, or *, or blank)
-13 = Bad interdigit timeout (must be non-negative)
-10X = Standard error codes (from dVR.h)

**************************************************************************/

void FAR vgettones ( 
  ParamBlk FAR *parm
 )
{
  int term_digits;  // number of digits to terminate on
  char term_key;   // key to terminate on
  int term_id_timeout; // interdigit timeout to terminate on in seconds
  int retries;     // number of retries
  char text[MAX_DIGITS + 1]; // digits buffer
  Value val;       // value struct for digits buf
  int nVRetVal;    // Voyys's return value
  Value result;   // FoxPro return structure
  struct line_t *sp; // ptr to slot in line table

  // find slot in the line table
  nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
  if ( nVRetVal != RC_SUCCEEDED )
    goto done;

  /*
  * Set the default values of the parameters here
  */
  term_digits = DEF_GETTONES_NUM_DIGITS;
  term_key = DEF_GETTONES_KEY_TO_TERMINATE;
  term_id_timeout = DEF_GETTONES_INTERDIGIT_TIMEOUT;
retries = 1;

/*
 * Check to see if any of the local overrides have been used.
 * If ANY of the three conditions has been used, this
 * overrides ALL of the defaults.
 */
if ( parm->pCount > 1 ) {
    term_digits = 0;
    term_key = ' ';
    term_id_timeout = 0;
}

/*
 * Now get the values of the parameters if they exist. Start
 * with the number of digits.
 */
if ( parm->pCount >= 2 ) {
    term_digits = ( ( int ) ( parm->p[1].val.ev_long ) );
}

/*
 * Now get the termination key (if any). This involves
 * resolving the handle to a string and copying the value
 * to the 'text' buffer.
 */
if ( parm->pCount >= 3 ) {
    _fstrncpy ( ( char far * ) text,
        _HandToPtr ( parm->p[2].val.ev_handle ),
        min ( MAX_DIGITS, parm->p[2].val.ev_length ) );
    term_key = text[0];
}

/*
 * Get the inter-digit timeout if any.
 */
if ( parm->pCount >= 4 ) {
    term_id_timeout = (int) parm->p[3].val.ev_long;
}

/*
 * Call get_tones with our arguments.
 */
nRetVal = com_Get_Tones ( 
    sp->hTask, 
    FALSE, 
    term_digits, 
    term_key, 
    term_id_timeout, 
    retries, 
    text,
    text
);

if (nRetVal == RC_DATALENGTH)
    nRetVal = RC_SUCCEED;
else if (nRetVal == RC_ENDFLAG)
    nRetVal = RC_TERMONSPECKEY;

/*
 * If the operation was successful, put the string in "text"
 * into the variable that was passed in.
 */
if ( nRetVal >= 0 ) {
/* Allocate space for the "Value" structure to be put into a variable, and for the text (handle). &&& Check for bad retcodes!

val.ev_type = 'C';
val.ev_handle = _AllocHand ( strlen ( text ) + 1 );
*/

/* Put the text value into the handle, and set the length in the value buffer.

*/
_StrCpy ( _HandToPtr ( val.ev_handle ), ( char far *)

15
val.ev_length = _StrLen ( ( char far *) text );

/*
* Now store the value into the variable.
*/
_Store ( ( Locator FAR * ) & ( parm->p0.loc ), &val );
}

done: // Send the return code back to FoxPro.

35
_MemFill ( &result, 0, sizeof ( result ) );
result.ev_type = 'I';
result.ev_long = nVRRetVal;
_RetVal ( &result );
return;

} /* function vgettones */

/******************************
Get touch tones (DTMF) macro: vgettonesm

retcode = vgettonesm( <voice_prompt> (C), <tries> (I),
<variable>(C) [,<number_of_digits>(I)]
[,<key_to_terminate_on>(C)]
[,<interdigit_timeout>(I)]
[,<valid_list>(C)]

vgettonesm is a "macro" combination of VSpeak and VGetTones.

It allows the programmer to specify a prompt, a number of retries, and a validation list in addition to the standard parameters of VGetTones.

The first parameter, which is mandatory, specifies a voice prompt to be spoken to the caller. The second parameter (also mandatory) specifies the total number of "tries" (including the first try) that are to be done.

The third (mandatory) parameter is a character variable to put the received keys (0-9, #, *) into. The next three parameters (optional) set the termination conditions. This can be one of more of:

- a set number of digits (0=any number of digits)
- a key to terminate on (like #) which is not to be included in the output ("*" = no key)
- an interdigit timeout (0=no interdigit timeout)
The absolute defaults are 1 keystroke, no terminating key, 10 second interdigit timeout. These defaults can be overridden by the use of the &seta; function. THOSE defaults, in turn, are overridden by any use of the optional arguments to this function.

The final parameter (optional) is a validation list. This list, in character form, contains all the valid choices, separated by commas.

Returns:
-2 = Terminated on interdigit timeout (success)
-1 = Terminated on specified key (success)
 0 = Terminated on number of digits (success)
-11 = Terminated on hangup (failure)
-12 = Bad termination key (must be 0-MAX_DIGITS)
-13 = Bad interdigit timeout (must be non-negative)
-10X = Standard error codes (from dvr.h)

/**
   * This procedure is not complete.
   */

int term_digits; // number of digits to terminate on
char term_key;  // key to terminate on
int term_id_timeout; // interdigit timeout to terminate on in seconds
int retries; // number of retries
char text[MAX_DIGITS + 1]; // digits buffer
Value val; // value struct for digits buf
int nVRetVal; // Voysys return value
Value result; // FoxPro return structure
struct line_t *sp; // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( GetCurrentTask(), &sp );
if ( nVRetVal != RC_SUCCEEDED )
   goto done;

/*
 * Set the default values of the parameters here
 */

term_digits = DEF_GETTIONS_NUM_DIGITS;
term_key = DEF_GETTIONS_KEY_TO_TERMINATE;
term_id_timeout = DEF_GETTIONS_INTERDIGIT_TIMEOUT;
retries = 1;

/*
 * Check to see if any of the local overrides have been used.
 * If ANY of the three conditions has been used, this overrides ALL of the defaults.
 */

if ( parm->pCount > 1 ) {
   term_digits = 0;
term_key = "";
term_id_timeout = 0;
}

/*
* Now get the values of the parameters if they exist. Start
* with the number of digits.
*/
if ( parm->pCount >= 2 ) {
    term_digits = (( int ) ( parm->p[1].val.ev_long ) ) ;
}

/*
* Now get the termination key (if any). This involves
* resolving the handle to a string and copying the value
* to the 'text' buffer.
*/
if ( parm->pCount >= 3 ) {
    _strncpy ( ( char far *) text,
               _HandToPtr ( parm->p[2].val.ev_handle ),
               min ( MAX_DIGITS, parm->p[2].val.ev_length ) ) ;
    term_key = text[0];
}

/*@ Get the inter-digit timeout if any. */
if ( parm->pCount >= 4 ) {
    term_id_timeout = (int) parm->p[3].val.ev_long;
}

/*@ Call get_tones with our arguments. */
NVRetVal = com_Get_Tones ( sp->hTask,
                          FALSE,
                          term_digits,
                          term_key,
                          term_id_timeout,
                          retries,
                          text );

if ( NVRetVal == RC_DATALENGTH)
    NVRetVal = RC_SUCCEEDED;
else if ( NVRetVal == RC_HNDFLAG)
    NVRetVal = RC_TERMONSPECKEY;

/*@ If the operation was successful, put the string in "text"
* into the variable that was passed in. */
if ( NVRetVal >= 0 ) {
    /*
    * Allocate space for the "Value" structure to be put into
    * variable,
    * and for the text (handle). &&& Check for bad retcodes!
    */
    val.ev_type = 'C';
    val.ev_handle = _AllocHand ( strlen ( text ) + 1 );
/*
 * Put the text value into the handle, and set the length
 * in the value buffer.
 */

_StrCpy (_HandToPtr (val.ev_handle), (char far *)
   text);
val.ev_length = _StrLen ((char far *) text);

/*
 * Now store the value into the variable.
 */
_Store ((Locator FAR *) & (parm->p[0].loc), &val);
}

done:
   // Send the return code back to FoxPro.
   _MemFill (&result, 0, sizeof (result));
   result.ev_type = 'I';
   result.ev_long = nRetVal;
   _RetVal (&result);
   return;
}

} /* function vgettonesm */

/*******************************************************************************/

Get words (voice recognition): VGETWORDS

retcode = VGETWORDS( @<variable>(C)
   [,<sub-vocabulary_name>(C)]
   [,<number_of_words>(I)]
   [,<word_to_terminate_on>(C)]
   [,<interword_timeout>(I)]
)

The first (mandatory) parameter is a character variable to put
the received words (0-9, oh, yes, no) into. The second (optional)
parameter specifies the sub-vocabulary name. Valid choices are:

   all   (default - the list of 0-9, oh, yes, no)
   numbers (0-9 and oh)
   yes/no (yes and no only)

The next three parameters (optional) set the termination
conditions. This can be one of more of:

   - a set number of words (0=any number of words)
   - a word to terminate on which is not to be included in
     the output (* " = no word)
   - an interword timeout (0=no interword timeout)

The absolute defaults are 1 word, no terminating word, no
interword timeout. These defaults can be overrided by the use
of another function. THOSE defaults, in turn, are overriden by
any use of the optional arguments to this function.

Returns:
   2   = Terminated on interword timeout (success)
   1   = Terminated on specified word (success)
   0   = Terminated on number of words (success)
   -1  = Terminated on timeout (failure)
   -2  = Terminated on hangup (failure)
- 98 -

-11 = Bad termination number of words (must be
  0-MAX_DIGITS)
-13 = Bad interword timeout (must be non-negative)
-10X = Standard error codes (from dVR.h)

******************************************************************************

void FAR vggetwords (  
  ParamBlk FAR *parm
)
{
  int   num_words;  /* number of words to terminate on */  
  char  term_word;  /* word to terminate on */
  ulong term_iw_timeout;  // interword timeout to terminate on
  int   retries;  /* number of retries */
  char  subvocab[10]; /* buffer for sub-vocab name & NEED
                        CONSTANT */
  char  text[MAX_DIGITS + 1];  /* digits buffer */
  Value val;  /* value struct for digits buf */
  int   nVRetVal;  // Voyys return value
  Value result;  // FoxPro return structure
  struct line_t *sp;  // ptr to slot in line table
  int   nCount;

  // find slot in the line table
  nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
  if ( nVRetVal != RC_SUCCEEDED )
    goto done;

  /*
  * Set the default values of the parameters & Need to
  * consult a table here
  */
  num_words = DEF_GETWORDS_NUM_WORDS;
  term_word = DEF_GETWORDS_WORD_TO_TERMINATE;
  term_iw_timeout = DEF_GETWORDS_INTERWORD_TIMEOUT;
  retries = 1;

  /* Get the sub-vocab name if one has been used. Otherwise,
  * use "ALL"
  */
  if ( parm->pCount >= 2 ) {
    _fstrncpy ( ( char far * ) subvocab,
               HandToPtr ( parm->p[1].val.ev_handle ),
               parm->p[1].val.ev_length );
    subvocab[parm->p[1].val.ev_length] = ( char ) NULL;
    trim_trailing ( subvocab );
   strupr ( subvocab );
  }
  else
    strcpy ( subvocab, "ALL" );

  /*
  * Check to see if any of the local overrides have been used.
  * If ANY of the three conditions has been used, this
  * overrides ALL of the defaults.
  */
  if ( parm->pCount > 2 ) {
    num_words = 0;
    term_word = ";
    term_iw_timeout = 10;  /* & KLUDGE */
  }
*/
* Now get the values of the parameters if they exist. Start
* with the number of words. Also check the values of the
* parameters as we go.
*/
if ( parm->pCount >= 3 ) {
    num_words = ( ( int ) ( parm->p[2].val.ev_long ) );
    if ( ( num_words < 0 ) || ( num_words > MAX_DIGITS ) ) {
        nVRetVal = -11;
        goto done;
    }
}

/*
* Now get the termination word (if any). This involves
* resolving the handle to a string and copying the value
* to the 'text' buffer.
*/
if ( parm->pCount >= 4 ) {
    nCount = min ( MAX_DIGITS, parm->p[3].val.ev_length );
    _strncpy ( ( char far *) text,
                HandToPtr ( parm->p[3].val.ev_handle ),
                nCount );
    text[nCount] = ( char ) NULL;
    trim_trailing ( text );
    term_word = text[0];
    if ( !strchr ( "0123456789oynOYN", term_word ) ) {
        Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vgetwords: bad
                     termination word" ) );
        nVRetVal = -12;
        goto done;
    }
}

/*
* Get the inter-word timeout if any.
*/
if ( parm->pCount >= 5 )
    term_iw_timeout = parm->p[4].val.ev_long;

/*
* Convert the timeouts to milliseconds.
*/
term_iw_timeout *= 1000;

/*
* Call get_words with our arguments.   &&& NOTE:
*   termination word not used!
*/
nVRetVal = com_Get_Words ( sp->hTask,
                           FALSE,
                           subvocab,
                           num_words,
                           term_iw_timeout,
                           retries,
                           text );
/*
 * If the operation was successful, put the string in "text"
 * into the variable that was passed in.
 */
if ( nVRetVal >= 0 ) {

    /*
    * Allocate space for the text (handle). & & Check for bad
    * retcode!
    */
    val.ev_type = 'C';
    val.ev_handle = _AllocHand ( strlen ( text ) + 1 );

    /*
    * Put the text value into the handle, and set the length
    * in the value buffer.
    */
    _StrCpy ( _HandToPtr ( val.ev_handle ), ( char far * )
        val.ev_length = _StrLen ( ( char far * ) text );

    /*
    * Now store the value into the variable.
    */
    _Store ( ( Locator FAR * ) & ( parm->p[0].loc ), &val );
}

done:
// Send the return code back to FoxPro.
    _MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = nVRetVal;
    _RetVal ( &result );
    Return;

} /* function vgetwords */

/******************************************************************************
Record: VRRECORD

retcode = VRRECORD( <string_or_variable>(C/R)
                           [,<max_recording_length>(I)] )

Play a beep and record a message from the caller. The first
parameter must be either a character string containing the file
name to be recorded to, or a character variable passed by
reference into which a unique file name will be placed.
The timeout and termination keypress will be gotten from the
defaults table.

Returns:
0 = Successful record
-1 = Failure - timeout
-11 = Invalid input (not a char or char variable)
-12 = Invalid file name
-13 = Invalid recording length; must be a positive integer
-10X = Standard error codes (from GVR.h)

******************************************************************************/
void FAR vrecord ( 
    ParamBlk FAR *parm 
) 
{
    char    filename[MAX_DN + MAX_FN];
    ulong   timeout;        // silence timeout value for record
    char    term_key;       // termination key for record
    Value   val;            // value struct for filename
    int     nVRetVal;       // Vsys return value
    Value   result;         // FoxPro return structure & used by
                            // MemoField operation too <<<<
    int     max_recording_len;    // maximum length of the
                                  // recording in seconds
    struct line_t *sp;        // ptr to slot in line table
    int     nCount;
    boolean bMemoField = FALSE;
    Locator loc;
    char    *pFileName;

    // find slot in the line table
    nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nVRetVal != RC_SUCCEED )
        goto done;

    /*
     * Check that the first parameter is a string, or a variable
     * passed by reference.
     */
    if ( parm->p[0].val.ev_type == 'C' && parm->p[0].loc.l_type != 'R' )
    {
        nVRetVal = RC_RECORDINVALIDINPUT;
        Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vrecord: invalid
                           input (not a character string or character variable passed by
                           reference)" ) );
    }
    goto done;
}

    /*
     * Set the defaults for the timeout and termination keys & &
     * Need to check a table here.
     */
    timeout = DEF_RECORD_TIMEOUT;
    term_key = DEF_RECORD_TERM_KEY;
    max_recording_len = DEF_RECORD_MAX_LENGTH;

    pFileName = filename;

    /*
     * Now get the file name into the proper buffer. If there is
     * no filename (call by reference), set a zero-length
     * string.
     */
    if ( parm->p[0].val.ev_type == 'C' )
    {
        nCount = max( MAX_FN, parm->p[0].val.ev_length );
        _fstrncpy ( ( char far *) filename, 
                    HandToPtr ( parm->p[0].val.ev_handle ),
                    nCount );
        filename[nCount] = ( char ) NULL;
        trim_trailing ( filename );
    }

    // try to find out if this is a memo field
    nVRetVal = IsMemoField ( filename, &bMemoField, &result,
if ( nVRetVal != RC_SUCCESS )
goto done;

if ( bMemoField ) {
    nVRetVal = GetTmpFilename ( sp->hTask, GET_RECORDINGS, filename );
    if ( nVRetVal != RC_SUCCESS )
goto done;

    // put data into temporary file filename
    // get the filename only
    pFileName = strrchr ( filename, '\');
    pFileName++;  // skip the '\' char
}
else {
    *filename = ( char ) NULL;
}

/* Get the "max_recording_length" parameter if there is one. */
if ( parm->pCount > 1 ) {
    max_recording_len = (( int ) parm->p[1].val.ev_long );
}

/* Call the 'record' function. */

/* & & timeout is not used ! ! ! */
nVRetVal = com_Record ( sp->hTask,
    pFileName,
    FALSE,
    term_key,
    timeout * 1000,
    max_recording_len );

if ( nVRetVal == 0 ) {
    /* successful record to filename

    * If the argument was a variable passed by reference, we
    * need to put "filename" back into it.
    */
    if ( parm->p[0].loc.l_type == 'R' ) {

    /* * Allocate space for the filename (handle). & & Check
    * for bad retcode! */
    val.ev_type = 'C';
    val.ev_handle = _AllocHand ( strlen ( filename ) + 1);  

    /* * Put the filename into the handle, and set the
    * length in the value buffer. */
    _StrCpy ( _HandToPtr ( val.ev_handle ), ( char far * )

    if ( parm->p[0].loc.l_type == 'R' ) {

    /* * Allocate space for the filename (handle). & & Check
    * for bad retcode! */
    val.ev_type = 'C';
    val.ev_handle = _AllocHand ( strlen ( filename ) + 1);  

    /* * Put the filename into the handle, and set the
    * length in the value buffer. */
    _StrCpy ( _HandToPtr ( val.ev_handle ), ( char far * )
val.ev_length = _StrLen ( ( char far *) filename );

/*
 * Now store the value into the variable.
 */
_STORE ( ( Locator FAR *) & ( parm->p[0].loc ), &val );
else if ( bMemoField ) {
    nVRetVal = GetMemoDataFromTmpFile ( filename, result,
    loc );
    if ( nVRetVal != RC_SUCCEED )
        goto finish;
    Debug ( 2, sprintf ( Debug_buf, "%DB_Fox/vrecord: fill
    memo field with data from file '%s', filename ) );
}

// always return success, no matter how the recording is done
nVRetVal = RC_SUCCEED;
}

finish:
if ( bMemoField ) {
    // delete the temporary file
    if ( remove ( filename ) < 0 ) {
        // may not need to return error code
        Debug ( 1, sprintf ( Debug_buf, "%DB_Fox/vrecord:
        cannot remove temporary file '%s', filename ) );
        goto done;
    }
}

done: // Send the return code back to FoxPro.
_MemFill ( &result, 0, sizeof ( result ) );
result.ev_type = 'I';
result.ev_long = nVRetVal;
_ReVal ( &result );
return;

} /* function vrecord */

ActionCode = VDIAL ( <phone_number>(C), <num_rings>(I) )
    [:<greeting_length>(R)] )

Go offhook, wait for dial tone, and dial the number passed (as a
character string). Only the characters: 0123456789*#ABCD will be
dialed; special chars allowed are:
T  use tone dialing (default)
P  use pulse dialing
W  wait for dial tone
,  pause 2 seconds
!  do a switch hook
The "rings" parameter indicates the number of rings to wait until deciding there is no answer. "rings" is optional, defaults to 5. If a "predictive dialing" effect is desired, the third parameter may be used. It must contain a numeric variable passed by reference. Upon exit, this variable will contain the length (in seconds) of the user's greeting.

Returns:
0 = Success
-3 = Busy
-4 = RNA
-9 = Other failure
-11 = Bad "rings" parameter; must be greater than 0
-10X = Standard error codes (from dVR.h)

*******************************************************************************/

void FAR vdialog ( ParamBlk FAR *parm

char text[MAX_DIGITS + 1]; /* text from Foxpro for digits */
char digits[MAX_DIGITS + 1]; /* digits buffer */
int rings; /* number of rings to wait */
boolean do_predictive; /* do predictive dialing? */
ulong greeting_length; /* if so, length of greeting in millisecs */
Value val; /* value struct for greeting len */
int nVRetVal; /* Vosys return value */
Value result; /* FoxPro return structure */
int count;
char *chrptr;
struct line_t *sp;
int RCount;

// find slot in the line table
nVRetVal = FindSlot ( GetCurrentTask ( ), &sp );
if ( nVRetVal != RC_SUCCEEDED )
goto done;

/*
 * Set the default values of the parameters & need to consult a table here
 * rings = DEF_DIAL_NUM_RINGS;
 */
do_predictive = FALSE;

/*
 * Get the dial string (first parameter) into "text", then convert it to upper-case.
 */
ncount = min ( MAX_DIGITS - 1, parm->p[0].val.ev_length );
_fstrncpy ( ( char far * ) text,
_HandToPtr ( parm->p[0].val.ev_handle ),
nCount );
text[nCount] = ( char ) NULL;
trim_trailing ( text );
strupr ( text );
/*
 * Start the digits string with a "W" (wait for ring). Then
 * move only those digits that are recognized into the
 * digits string.
 */
digits[0] = 'W';
count = 1;
chrptr = text;
while ( chrptr = strpbrk ( chrptr, "0123456789*#ABCDFPW,*" ) )
    digits[count++] = *chrptr++;
digits[count] = ( char ) NULL;

/*
 * Get the "rings" parameter if there is one.
 */
if ( parm->pCount > 1 ) {
    rings = ( ( int ) parm->p[1].val.ev_long );
}

/*
 * Check if there is a "predictive" parameter.
 */
if ( parm->pCount > 2 )
    do_predictive = TRUE;

/*
 * Call the compute-layer routine.
 */
qVRetVal = com_CallOut ( sp->hTask, digits, rings,
    do_predictive, &greeting_length );

/*
 * If the operation was successful, and predictive dialing was
 * on, we want to put the result of the operation back
 * into the variable passed in by reference.
 */
if ( do_predictive && ( qVRetVal >= 0 ) ) {
    val.ev_type = 'I';

    /*
    * Put the greeting length into the structure (dividing by
    * 1000, rounded up, to convert the milliseconds to
    * seconds).
    */
    val.ev_long = ( greeting_length + 500 ) / 1000;

    /*
    * Now store the value into the variable.
    */
    _Store ( ( Locator FAR * ) & ( parm->p[2].loc ), &val );
}

done:
    // Send the return code back to FoxPro.
    MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = qVRetVal;
   RetVal ( &result );
    Return;
} /* function vdial */
Set Debug Level: VDEBUG

```c
retcode = VDEBUG(<level>(I))
```

Debug levels:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No debug printouts</td>
</tr>
<tr>
<td>1</td>
<td>Standard debug printouts</td>
</tr>
<tr>
<td>2</td>
<td>Extended debug printouts</td>
</tr>
</tbody>
</table>

Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
</tbody>
</table>

Set the directories for voice files: VSETDIR

```c
retcode = VSZETDIR(<voice_file_type>(C), <directory>(C))
```
Valid voice file types:
    SysPrompts  (old form: sys_prompts)  system prompts
    AppPrompts  (old form: app_prompts)  application prompts
    Recordings  user recordings

5

Returns:
   0    = Success
   -71   = failure: invalid voice file type

void FAR vsetdir ( 
    ParamBlk FAR *parm
)
{
    int nRetVal;  // Vcysys return value
    Value result;  // FoxPro return structure
    char dir_type[MAX_DN + 1];
    char text[MAX_DN + 1];
    struct line_t *sp;  // ptr to slot in line table
    int nCount;
    char szLibraryName[MAX_IN + 1];

    // find slot in the line table
    nRetVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nRetVal != RC_SUCCEED )
        goto done;

    // Note: cannot call this in start as the library is not
    // finished loading so the library name is not available yet
    GetLibraryName( szLibraryName );

    /*
    * Get the text of the first argument.
    */
    nCount = min ( MAX_DN, parm->p[0].val.ev_length );
    _fstrncpy ( ( char far *) dir_type,
                _HandToPtr ( parm->p[0].val.ev_handle ),
                nCount );
    dir_type[nCount] = ( char ) NULL;
    trim_trailing ( dir_type );

    /*
    * Get the text of the second argument, and capitalize it.
    */
    _fstrncpy ( ( char far *) text,
                _HandToPtr ( parm->p[1].val.ev_handle ),
                parm->p[1].val.ev_length );
    text[parm->p[1].val.ev_length] = ( char ) NULL;
    trim_trailing ( text );

    /*
    * Call com_Set_Directory to set the directory.
    */
    nRetVal = com_Set_Directory ( sp->hTask, dir_type, text );

done:
    // Send the return code back to FoxPro.
    _MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = nRetVal;
    _RetVal ( &result );
return;
}  /* function vsetdir */

/***************************************************************************/
 Set the speak mode for dates: VSetDateSpeak

retcode = VSetDateSpeak ("mmdyyyy" | "mmd" | "ddmmyy" | "dmm"
)

Returns:
  0  = Success
-72  = failure: invalid date speak type

/***************************************************************************/

void FAR vsetdatespeak ( 
  ParamBlk FAR *parm
)
{
  int nVRetVal;        // VoySys return value
  Value result;        // FoxPro return structure
  char text[MAX_DN + 1];  // ptr to slot in line table
  struct line_t *sp;
  nCount;

  // find slot in the line table
  nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
  if ( nVRetVal != RC_SUCCESS )
    goto done;

  /*
   * Get the text of the first argument.
   */
  nCount = min( MAX_DN, parm->p[0].val.ev_length );
  _fstrncpy ( ( char far * ) text,
              _HandToPtr ( parm->p[0].val.ev_handle ),
              nCount );
  text[nCount] = ( char ) NULL;
  trim_trailing ( text );

  /*
   * Set the global speak type
   */
  nVRetVal = com_Set_Date_Speak ( sp->hTask, text );

done:  // Send the return code back to FoxPro.
  _MemFill ( &result, 0, sizeof ( result ) );
  result.ev_type = 'I';
  result.ev_long = nVRetVal;
  _RetVal ( &result );
  return;
}  /* function vsetdatespeak */
Set default timeouts: VSetTimeout

```
    retcode = VSetTimeout("VGetTones" | "VGetWords, <timeout>")

    Sets the default first-key/first-word timeouts for GetTones and GetWords.

    Returns:
    - 0 = Success
    -76 = failure: invalid keyword (first argument)
    -77 = failure: invalid timeout (must be positive integer)
```

void FAR vsSettimeout ( 
    ParamBlk FAR *parm
)
{
    int nVRetVal;        // Voysys return value
    Value result;       // FoxPro return structure
    int timeout;
    char text[MAX_DN + 1]; // ptr to slot in line table
    int nCount;

    // find slot in the line table
    nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nVRetVal != RC_SUCCEEDED )
        goto done;

    /* Get the text of the first argument. */
    nCount = min( MAX_DN, parm->p[0].val.ev_length );
    _fstrncpy ( ( char far * ) text,
                _HandleToPtr ( parm->p[0].val.ev_handle ),
                nCount );
    text[nCount] = ( char ) NULL;
    trim_trailing ( text );

    /*
    * Get the timeout value
    */
    timeout = (int) parm->p[1].val.ev_long;

    nVRetVal = com_Set_Timeout ( sp->hTask, text, timeout );

done:
    // Send the return code back to FoxPro.
    _MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = nVRetVal;
    _RetVal ( &result );
    return;
}
/* function vsSettimeout */
- 110 -

/***************************************************************************/
/*
Setup Fax Cover Page : VFaxCover

retcode = VFaxCover( C, <string_or_variable>(C/R) )

Specify the cover page being used for faxes.

Returns:
 0      = Success
-11    = Invalid input
-10X   = Standard error codes (from dVR.h)

***************************************************************************/

void FAR vfaxcover ( 
  ParamBlk FAR *parm)
{
  char cText[MAX_FAXCN + 1];
  char cParm[MAX_FAXPARAM + 1]; // Buffer for file name
  int nVRetVal; // Voyysys return value
  Value result; // FoxPro return structure
  int nFaxCoverCmdType; // fax cover page command
  int nCount;  // temporary counter
  LPOVOID lpParm;  // command parameter pointer
  struct line_t *sp; // ptr to slot in line table

  // find slot in the line table
  nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
  if ( nVRetVal != RC_SUCCEEDED )
    goto done;

  // Verify if the first parameter is of proper type
  if ( parm->p[0].val.ev_type != 'C' ) {
    nVRetVal = -11;
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxcover: invalid input (not a character string)" ) );
    goto done;
  }

  // Get the text of the first argument and make the string uppercase.
  // uppercas.
  nCount = min( MAX_FAXCN, parm->p[0].val.ev_length );
  _strncpy ( (char far *) cText, HandToPtr( parm->p[0].val.ev_handle ), nCount );
  cText[nCount] = ( char ) NULL;
  trim_trailing ( cText );
 strupr ( cText );

  // Parse the fax cover page setup command.
  if ( (!_strcmp ( cText, "NEW" ) )
    nFaxCoverCmdType = FAX_COVER_NEW;
  else if ( (!_strcmp ( cText, "NOTEXT" ) )
    nFaxCoverCmdType = FAX_COVER_NOTEXT;
  else if ( (!_strcmp ( cText, "LOGOTOP" ) )
    nFaxCoverCmdType = FAX_COVER_LOGOTOP;
  else if ( (!_strcmp ( cText, "LOGOBOTTOM" ) )
    nFaxCoverCmdType = FAX_COVER_LOGOBOTTOM;
  else if ( (!_strcmp ( cText, "RENAME" ) )
    nFaxCoverCmdType = FAX_COVER_RENAME;
  nFaxCoverCmdType = FAX_COVER_RENAME;
else if (!strcmp ( cText, "RECCONAME" ))
    nFaxCoverCmdType = FAX_COVER_RECCONAME;
else if (!strcmp ( cText, "SENDERNAME" ))
    nFaxCoverCmdType = FAX_COVER_SENDERNAME;
else if (!strcmp ( cText, "SENDERPHONE" ))
    nFaxCoverCmdType = FAX_COVER_SENDERPHONE;
else if (!strcmp ( cText, "RECCONAME" ))
    nFaxCoverCmdType = FAX_COVER_RECCONAME;
else if (!strcmp ( cText, "COVERTEXT" ))
    nFaxCoverCmdType = FAX_COVER_COVERTEXT;
else if (!strcmp ( cText, "COVERTEXTFILE" ))
    nFaxCoverCmdType = FAX_COVER_COVERTEXTFILE;

    // Get 2nd parameter if there need to be one
    switch ( nFaxCoverCmdType ) {
        case FAX_COVER_NEW: 
        case FAX_COVER_NOTEXT: 
            lpParm = 0;
            break;
        case FAX_COVER_LOGOTOP: 
        case FAX_COVER_LOGOBOTTOM: 
        case FAX_COVER_COVERTEXTFILE: 
        case FAX_COVER_RECCONAME: 
        case FAX_COVER_SENDERPHONE: 
        case FAX_COVER_SENDERCONAME: 
        case FAX_COVER_COVERTEXT: 
            if ( parm->p[1].val.ev_type == 'C' ) {
                nCount = min(MAX_FAXPARM, parm->p[1].val.ev_length);
                _fstrncpy ( (char far *) cParm, _HandToPtr (parm->p[1].val.ev_handle ), nCount );
                cParm[nCount] = (char) NULL;
                trim_trailing ( cParm );
                lpParm = (void far *)cParm;
            } else {
                nVRetVal = -11;
                Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxcover: invalid second parameter" ));
                goto done;
            }
            break;
        default: 
            nVRetVal = -11;
            Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxcover: invalid command" ));
            goto done;
    }

    // Call the faxcover function.
    nVRetVal = com_Faxcover (sp->hTask, nFaxCoverCmdType, lpParm);
    if ( nVRetVal == 0 )
        Debug ( 3, sprintf ( Debug_buf, "DB_Fox/vfaxcover: successful set cover page" ));

    done:
    // Send the return code back to FoxPro.
    _MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = nVRetVal;
_RetVal ( &result );
    return;
} /* function vfaxcover */

/*********************************************************
Add pages to fax document to be sent: VFaxDoc
retCode = VFaxDoc( C )
Specify the fax pages being used for faxes.
Returns:
  0   = Success
-11  = Invalid input
-10X = Standard error codes (from dvr.h)
*********************************************************/
void FAR vfaxdoc ( 
    ParamBlk FAR *parm
)
{
    char cPageName[MAX_FN +1 ]; // Buffer for file name
    int nRVal;
    int result;
    int nFaxDocCmdType;
    int nCount;
    LPVOID lpParm;
    struct line_t *sp;

    // find slot in the line table
    nRVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nRVal != RC_SUCCEEDED )
        goto done;

    // Verify if the first parameter is of proper type
    if ( parm->p[0].val.ev_type != 'C' )
        goto done;

    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxdoc: invalid input (not a character string)" ) );

    // Get the text of the first argument and make the string uppercase.
    nCount = min( MAX_FN, parm->p[0].val.ev_length );
    _strncpy ( (char far *) cPageName, _ HankToPtr( parm->p[0].val.ev_handle ), nCount );
    cPageName[nCount] = ( char ) NULL;
    trim_trailing ( cPageName );
   strupr ( cPageName );

    // Parse the fax cover page setup command.
    if ( !strcmp ( cPageName, "NEW" ) )
        nFaxDocCmdType = FAX_DOC_NEW;
    else if ( !strcmp ( cPageName, "PAGEBREAK" ) )
        nFaxDocCmdType = FAX_DOC_PAGEBREAK;
lpParm = 0;
}
else {
    nFaxDocCmdType = FAX_DOC_FILE_BY_EXT;
    lpParm = cPageName;
}

// Call the fax doc function.
if ( nRetVal >= 0 )
    Debug ( 3, sprintf ( Debug_buf, "DB_Fox/vfaxdoc: successfully set document page" ) );

done:
    // Send the return code back to FoxPro.
    _MemFill ( &result, 0, sizeof ( result ) );
    result.ev_type = 'I';
    result.ev_long = nRetVal;
    _RetVal ( &result );
    Return;

} /* function vfaxdoc */

/*********************************************************************************/

Send fax document to the specified number : VFaxSend

retcode = VFaxSend ( c [, C ] )

Send the active fax document or the specified document.

Returns:
    0      = Success
    -11    = Invalid input
    -10X   = Standard error codes (from dVR.h)

*********************************************************************************/

void FAR vfaxsend ( 
    ParamBlk FAR *parm
)
{
    char cDocName[MAX_FN + 1];    // Buffer for file name
    char cDialStr[MAXDIGITS + 1]; // Buffer for dial string
    int nRetVal;                 // Voysys return value
    Value result;                // FoxPro return structure
    int nFaxSendCmdType;         // fax send command
    int nCount;                  // temporary counter
    LPVOID lpParm;               // command parameter pointer
    struct line_t *sp;           // ptr to slot in line table

    // find slot in the line table
    nRetVal = FindSlot ( GetCurrentTask ( ), &sp );
    if ( nRetVal != RC_SUCCEEDED )
        goto done;

    // Verify if the first parameter is of proper type
    if ( parm->p[0].val.ev_type != 'C' ) {
        nRetVal = -11;
        return;
    }
Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxsend: invalid
input (not a character string)" ) );
goto done;
}

// Get the text of the first argument and make the string
// uppercase.
ncount = min ( MAX_DIGITS, parm->p[0].val.ev_length );
_fstrncpy ( (char far *)cDialStr, _HandToPtr( 
parm->p[0].val.ev_handle ), nCount );
cDialStr[ncount] = ( char ) NULL;
trim_trailing ( cDialStr );
strupr ( cDialStr );

// Get the optional document file name
if ( parm->pcount == 1 ) {
    nFaxSendCmdType = FAX_SEND_ACTIVE_DOC;
lParm = 0;
}
else {
    nCount = min ( MAX_FN, parm->p[1].val.ev_length );
    _fstrncpy ( (char far *)cDocName, _HandToPtr( 
parm->p[1].val.ev_handle ), nCount );
cDocName[ncount] = ( char ) NULL;
trim_trailing ( cDocName );
strupr ( cDocName );
nFaxSendCmdType = FAX_SEND_FILE_BY_EXT;
lParm = cDocName;
}

// Call the fax send function.

nRetVal = com_FaxSend ( sp->hTask, cDialStr, nFaxSendCmdType,
lParm );

if ( nRetVal >= 0 )
    Debug ( 3, sprintf ( Debug_buf, "DB_Fox/vfaxsend:
  successfully queue fax document for sending out" ) );

done:
// Send the return code back to FoxPro.
_MemFill ( &result, 0, sizeof ( result ) );
result.ev_type = 'I';
result.ev_long = nRetVal;
_ResetVal ( &result );
Return;

} /* function vfaxsend */

/**********************************************************************
Setup fax options and parameters : VFaxSetup

  retcode = VFaxSetup( C [,?] )

  Setup fax options such as fax channel, send time, retry
  counts, etc.

Returns:
  0    = Success
-11   = Invalid input
-10X  = Standard error codes (from dVR.h)
void FAR vfaxsetup ( 
    ParamBlk FAR *parm 
) 
{
    char cOptName[MAX_FAXCN +1 ]; // buffer for option name
    char cOptVal[MAX_FAXPARM + 1]; // buffer for string option value
    int nOptVal; // integer option value
    int nRetVal; // Vysys return value
    Value result; // FoxPro return structure
    int nFaxSetupCmdType; // fax send command
    int nCount; // temporary counter
    LPVOID lpParm; // command parameter pointer
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nRetVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nRetVal != RC_SUCCEEDED )
        goto done;

    // Verify if the first parameter is of proper type
    if ( ((parm->p[0].val.ev_type != 'C') || (parm->pCount != 2)) )
        nRetVal = -11;
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxsetup: invalid input" ) );
    goto done;
}

// Get the text of the option name and make the string uppercase.
// uppercase.
nCount = min( MAX_FAXCN, parm->p[0].val.ev_length );
_fstrncpy ( (char FAR *)&OptName, HandToPtr( 
    parm->p[0].val.ev_handle ), nCount );
cOptName[nCount] = (char) NULL;
trim_trailing ( cOptName );
strupr ( cOptName );

// Convert the option name to option code
if ( (!strcmp( cOptName, "TRKS" ) )
    nFaxSetupCmdType = FAX_SETUP_TRYCOUNT; 
else if ( (!strcmp( cOptName, "TRYINTERVAL" ) )
    nFaxSetupCmdType = FAX_SETUP_TRYINTERVAL; 
else if ( (!strcmp( cOptName, "TIMEFROM" ) )
    nFaxSetupCmdType = FAX_SETUP_TIMEFROM; 
else if ( (!strcmp( cOptName, "TIMETO" ) )
    nFaxSetupCmdType = FAX_SETUP_TIMETO; 
else if ( (!strcmp( cOptName, "FAXMODE" ) )
    nFaxSetupCmdType = FAX_SETUP_FAXMODE; 
else if ( (!strcmp( cOptName, "FAXSIZE" ) )
    nFaxSetupCmdType = FAX_SETUP_FAXSIZE; 
else if ( (!strcmp( cOptName, "STATIONID" ) )
    nFaxSetupCmdType = FAX_SETUP_STATIONID; 
else if ( (!strcmp( cOptName, "FAXCHANNEL" ) )
    nFaxSetupCmdType = FAX_SETUP_FAXCHANNEL; 
else if ( (!strcmp( cOptName, "SENDHEADER" ) )
    nFaxSetupCmdType = FAX_SETUP_SENDHEADER; 
else {
    nRetVal = -11;
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vfaxsetup: invalid input (Unknown Option)" ) );
}
goto done;
}

// Get the option value according to the option type
switch ( nFaxSetupCmdType ) {
    case FAX_SETUP_TRYCOUNT: 
        case FAX_SETUP_TRYINTERVAL:
        case FAX_SETUP_FAXCHANNEL: 
            if ( parm->p[1].val.ev_type != 'I' ) {
                nRetVal = -11;
                Debug ( 1, printf ( Debug_buf, "DB_Fox/vfaxsetup:
                        invalid input (option must be INT)" ) );
                goto done;
            }
            nOptVal = (int)(parm->p[1].val.ev_long);
            lpParm = 0;
            break;
    case FAX_SETUP_TIMEFROM: 
    case FAX_SETUP_TIMETO: 
    case FAX_SETUP_FINEMODE: 
    case FAX_SETUP_PAGESIZE: 
    case FAX_SETUP_STATIONID: 
    case FAX_SETUP_SENDHEADER: 
        if ( parm->p[1].val.ev_type != 'C' ) {
            nRetVal = -11;
            Debug ( 1, printf ( Debug_buf, "DB_Fox/vfaxsetup:
                        invalid input (option must be CHAR)" ) );
            goto done;
        }
        nCount = min ( MAX_FAXPARAM, parm->p[1].val.ev_length );
        _fstrncpy ( (char far *)coOptVal, _HandToPtr(
                        parm->p[1].val.ev_handle ), nCount );
        coOptVal[nCount] = (char)NULL;
        trim_trailing ( coOptVal);
       strupr ( coOptVal );
        lpParm = coOptVal;
        nOptVal = 0;
        break;
    default: 
        // Impossible!
        break;
}

// Call the fax setup function.

nRetVal = com_FaxSetup ( sp->hTask, nFaxSetupCmdType, lpParm, 
                        nOptVal);

if ( nRetVal >= 0 )
    Debug ( 3, printf ( Debug_buf, "DB_Fox/vfaxsetup:
                        successfully setup fax options" ) );

done:
   // Send the return code back to FoxPro.
   _MemFill ( &result, 0, sizeof ( result ) );
   result.ev_type = 'Y';
   result.ev_long = nRetVal;
   _RetVal ( &result );
   return;

} /* function vfaxsetup */
#endif /* DO_FAX */

/*============================================================================
5 Set options and parameters : VSet

    retcode = VSet( C [,?] )

    Set options such as test mode, etc.

10 Returns:
    0 = Success
    -11 = Invalid input
    -10X = Standard error codes (from dVR.h)

/*============================================================================/

void FAR vset ( ParamBlk FAR *parm
20 )
{
    char      cOptName[MAX_SETCN + 1]; // buffer for option name
    char      cOptVal[MAX_SETPARM + 1]; // buffer for string
    ulong     dwOptVal; // unsigned long option value
    int       nVRetVal; // Voyysys return value
    Value     result; // FoxPro return structure
    int       nSetCmdType; // set command
    int       nCount; // temporary counter
    LPVOID    lpParm; // command parameter pointer
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( GetCurrentTask ( ), &sp );
    if ( nVRetVal != RC_SUCCEED )
        goto done;

    // Verify if the first parameter is of proper type
    if ( (parm->p[0].val.ev_type != 'C') || (parm->pCount != 2) )
    {
        nVRetVal = -11;
        Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vset: invalid
                                     input" ) );
        goto done;
    }

    // Get the text of the option name and make the string
    // uppercase.
    nCount = min( MAX_SETCN, parm->p[0].val.ev_length );
    _fstrncpy ( (char far *)cOptName, _handToPtr( 25
                                          parm->p[0].val.ev_handle ), nCount );
    cOptName[nCount] = (char) NULL;
    trim_trailing ( cOptName );
   strupr ( cOptName );

    /* Note: to support a new option, add the defines for command
     type and option value in command.h also you may use some
     of the defines in compute.h (or add new ones) for option
     values (e.g. for DateSpeak - DS_MMDDYY, DS_MMDD, etc.) */

35    // Convert the option name to option code
    if ( !strcmp ( cOptName, "TESTMODE" ) )
        nSetCmdType =
30
          SET_TESTMODE;

else {
    nVRetVal = -11;
    Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vset: invalid input (Unknown Option)" ));
    goto done;
}

// Get the option value according to the option type
switch ( nSetCmdType ) {
  case SET_TESTMODE:
    if ( parm->p[1].val.ev.type != 'C' ) {  // need a new error code (ML)
      nVRetVal = -11; // invalid input (option must be CHAR)
      Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vset: invalid input (option must be CHAR)" ));
      goto done;
    }
    nCount = min( MAX_SETPARM, parm->p[1].val.ev.length );
    _fstrncpy ( (char far *)cOptVal, _HandToPtr( parm->p[1].val.ev.handle ), nCount );
    cOptVal[nCount] = ( char ) NULL;
    trim_trailing ( cOptVal );
    strupr ( cOptVal );

    /*
    * Check through the list of allowable test mode types. Return an error if the user specified
    * an illegal test mode type.
    */
    if ( !strcmp ( cOptVal, "ON" ) ) {
      dwOptVal = SET_TM_ON;
    } else if ( !strcmp ( cOptVal, "OFF" ) ) {
      dwOptVal = SET_TM_OFF;
    } else {
      nVRetVal = -81;  // error code
      Debug ( 1, sprintf ( Debug_buf, "DB_Fox/vset: Illegal test mode type >%s<", cOptVal ) );
      goto done;
    }
    lpParm = 0;
    break;

default:
  // Impossible!
  break;
}

// Call the set function
nVRetVal = com_Set ( sp->hTask, nSetCmdType, lpParm, dwOptVal);

if ( nVRetVal >= 0 )
  Debug ( 3, sprintf ( Debug_buf, "DB_Fox/vset: successfully set options" ) );

done:
  // Send the return code back to FoxPro.
  _MemFill ( &result, 0, sizeof ( result ) );
  Result.ev_type = 'I';
  result.ev_long = nVRetVal;
- 119 -

RetVal ( &result );
return;
}

} /* function vset */

/***********************************************************
** Linkage information for FoxPro.**
****************************************************************/

/* &&& this should be static; next one can't be */
/* function names (the 1st column of data) must be in all */
/* capitals */

FoxInfo myFoxInfo[] = {
  "START", ( FFPi ) start, CALLONLOAD, "";
  "STOP", ( FFPi ) stop, CALLONUNLOAD, "";
  "VWAITRING", ( FFPi ) vwaiting, 2, ".I,.I*",
  "VHANGUP", ( FFPi ) vhangup, 0, "*",
  "VCHKHANGUP", ( FFPi ) vchkhangup, 0, "*",
  "VSPK", ( FFPi ) vspeak, 10,
  ".?.?..?,?..?,?..?,?..?,?..?,?..?*,
  "VGETTONES", ( FFPi ) vgettones, 4, "R,.I,.C,.I*",
  "VGETTONESM", ( FFPi ) vgettonesm, 7,
  "C, I, R,.I,.C,.I,.C" ,
  "VGETWORDS", ( FFPi ) vgetwords, 5, "R,.C,.I,.C,.I*",
  "VRRECORD", ( FFPi ) vrecord, 2, "?,.I*",
  "VDIAL", ( FFPi ) vdial, 3, "C,.I,.R*",
  "VDEBUG", ( FFPi ) vdebug, 1, "I*",
  "VSETDIR", ( FFPi ) vsetdir, 2, "C,C*",
  "VSETDSPK", ( FFPi ) vsetdatespeak, 1, "C*",
  "VSETTMOUT", ( FFPi ) vsetttimeout, 2, "C, I*",
  "VFAXCOVER", ( FFPi ) vfaxcover, 2, "C,.?*",
  "VFAXDOC", ( FFPi ) vfaxdoc, 1, "C*",
  "VFAXSEND", ( FFPi ) vfaxsend, 2, "C,.C*",
  "VFAXSETUP", ( FFPi ) vfaxsetup, 2, "C,.?*",
  "VSET", ( FFPi ) vset, 2, "C,.?*",
};

FoxTable _FoxTable = {
  ( FoxTable FAR * ) 0,
  sizeof ( myFoxInfo ) / sizeof ( FoxInfo ),
  myFoxInfo
};

/***********************************************************/
APPENDIX D
Compute.c
Copyright 1992-1994 Voyaya Corporation

FILE: Compute.c

Purpose:
Computation layer.
Does generic stuff needed for most applications, such as
translating numbers and dates into play-lists.
Independent of OS, line card type.

$Log: /VoyayaAccess/core/compute/COMPUTE.C $

*******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// if define equals to 1, enable fax support
// need to define this before the .h files as it could be used in
// the .h header files
#define DO_FAX  0

#include "OS.h"
#include "dVR.h"
#include "Command.h"
#include "Compute.h"
#include "IPC.h"

#define BASEDIALTIME_W 3000 // 3 seconds
#define BASEDIALTIME_COMMA 2000 // 2 seconds
#define BASEDIALTIME_DEFAULT 200 // .2 seconds

#define MAX_chrBUF 20

// list of prompt files defines
#define PF_0 "0"
#define PF_1 "1"
#define PF_2 "2"
#define PF_3 "3"
#define PF_4 "4"
#define PF_5 "5"
#define PF_6 "6"
#define PF_7 "7"
#define PF_8 "8"
#define PF_9 "9"
#define PF_MINUS "minus"
#define PF_MINUS "sign3"
#define PF_POINT "point"
#define PF_HUNDRED "hundred"
#define PF_BILLION "billion"
- 121 -

#define PF_MILLION "million"
#define PF_THOUSAND "thousand"
#define PF_DOLLAR "dollard"
#define PF_DOLLARS "dollars"
#define PF_CENT "cent"
#define PF_CENTS "cents"

#define PF_THE "the"
#define PF_OF "of"
#define PF_1990 "1990"
#define PF_1994 "1994"
#define PF_1995 "1995"
#define PF_1996 "1996"
#define PF_1997 "1997"
#define PF_1998 "1998"
#define PF_1999 "1999"
#define PF_2000 "2000"
#define PF_19 "19"

/*
 * Information about state of each active instance.
 * This data is shared by all client processes, but each process
 * looks only at the data for itself.
 * Processes find a free slot by looking at the bInUse field.
 */
struct line_t {

    boolean bInUse; // is this table slot in use?
    unsigned int hTask; // Windows task handle
    int nLineNum; // line number
    boolean bConnected; // connected to a line ?
    boolean bOnLine; // com_answer called ?
    int nDateSpeakType; // Current Set_Date_Speak

    /* Start (first key/first word) timeouts for GetTones and*/
    /* GetWords, in milliseconds */
    ulong lGetTonesStartTimeout;
    ulong lGetWordsStartTimeout;

    short nNumItemsInPlayList; // number of items added
to play list (used by com_play)

};

static line_t line[MAX_LINES] = {0};

/*********************************************/
// Find line table slot that is being used by task handle hTask

static
int FindSlot (
const unsigned int hTask; // task handle (Windows only)
struct line_t **sp; // ptr to ptr to slot in line table

int nRetVal; // Voyys return value
int nSlot; // slot in line table

// find slot in the line table by using hTask
for ( nSlot = 0 ; nSlot < MAX_LINES ; nSlot++ ) {
    if ( line[nSlot].bInUse && ( line[nSlot].hTask == hTask ) )
        break;
}

if ( nSlot >= MAX_LINES ) {
    nRetVal = RC_NOTINITIALIZED; // best guess
goto done;
}

*sp = &line[nSlot];
nRetVal = RC_SUCCEEDED;

done:
    return nRetVal;
}

/* function FindSlot */

/*******************
static
int add_voice (
    const struct line_t *sp; // ptr to slot in line table
    const char *voice_file,
    const int path[MAX_DIRS])
{
    int nRetVal; // Voyys return value
    struct command cmd; // command to send
    struct event event; // event received
    int i;

    Debug ( 4, sprintf ( Debug_buf, "Compute/add_voice: called, voice_file '%s', voice_file ");

    if ( strlen ( voice_file ) >= ( MAX_DN + MAX_FN ) ) {
        nRetVal = RC_BADFILENAME;
        goto done;
    }

    cmd.nCommand = DVRRC_ADDPLAY;
    cmd.nLine = sp->nLineNum;
    for ( i = 0 ; i < MAX_DIRS ; i++ )
        cmd.path[i] = path[i];
    cmd.play_item.type = PI_SINGLE_FILE;
    strcpy ( cmd.play_item.fname, voice_file );
    strcpy ( cmd.play_item.index = -1;
    cmd.play_item.vdata = ( void * ) 0xFFFFFFFF;

    nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
done:
    Debug ( 4, sprintf ( Debug_buf, "Compute/add_voice: ret %d", nRetVal );
    return ( nRetVal );
}

// function add_voice */

/***************************************************************************/
static int addp (
    const struct line_t *sp, // ptr to slot in line table
    const char *voice_file
)
{
    int nRetVal;
    int path[MAX_DIRS];
    int i;

    path[0] = DIR_SYSPROMPTS;
    for ( i = 1; i < MAX_DIRS; i++ )
        path[i] = DIR_END;

    nRetVal = add_voice ( sp, voice_file, path );

    return ( nRetVal );
}

// function addp */

/***************************************************************************/
static
int addu ( 
    const struct line_t *sp, // ptr to slot in line table
    const char *voice_file
)
{
    int nRetVal;
    int path[MAX_DIRS];
    int i;

    // Note: remember to add more DIR_ if new ones have been created

    path[0] = DIR_APPPROMPTS;
    path[1] = DIR_USERVOICE;
    path[2] = DIR_TMPFILES;
    for ( i = 3; i < MAX_DIRS; i++ )
        path[i] = DIR_END;

    nRetVal = add_voice ( sp, voice_file, path );

    return ( nRetVal );
}

} /* function addu */

/***************************************************************************/

The "speak as" routines assume the existence of the following prompts:
- 124 -

1 10 11 hundred
2 20 12 thousand
3 30 13 million billion
4 40 14 the sign3 (minus)
5 50 15 point of
6 60 16 dollar dolland ("dollar and")
7 70 17 dollars dollands ("dollars and")
8 80 18 cent
cents
9 90 19

10

monday tuesday wednesday thursday friday saturday
sunday

15
january february march april
may june july august
september october november december

20
1st 2nd 3rd 4th 5th 6th 7th 8th 9th
10th 11th 12th 13th 14th 15th 16th 17th 18th
19th 20th 21st 22nd 23rd 24th 25th 26th 27th
28th 29th 30th 31st

******************************************************************************

25
/
* Special prompt lists. Remember that in C, the first element
* in an array has the index 0.
*/

30 static const
char *Days[] = {
"sunday",
"monday",
"tuesday",
"wednesday",
"thursday",
"friday",
"saturday"
};

40 static const
char *Months[] = {
"january",
"february",
"march",
"april",
"may",
"june",
"july",
"august",
"september",
"october",
"november",
"december"
};

55 static const
char *Ordinals[] = {
"1st", "2nd", "3rd", "4th", "5th",
"6th", "7th", "8th", "9th", "10th",
"11th", "12th", "13th", "14th", "15th",
"16th", "17th", "18th", "19th", "20th",
};
"21st", "22nd", "23rd", "24th", "25th",
"26th", "27th", "28th", "29th", "30th",
"31st"
});

/*****************

speak_as_digits: Take the character input, and speak out the
different digit prompts. If preceded with a minus, speak
that. Also allow for decimal points and digits after that.
Ignore all characters other than '0'-'9', '-', and '.'.
This procedure can handle any number of digits. This
procedure will ignore any characters that are not digits.
returns: 0 = success
******************/

static int speak_as_digits ( const struct line_t *sp,
    const char *nmbuf
)
{
    int nVRetVal;
    const char *chrptr;
    chrptr = nmbuf;

    while (*chrptr != (char) NULL)
    {
        switch (*chrptr )
        {
            case '0':
                nVRetVal = addp ( sp, PF_0 );
                break;
            case '1':
                nVRetVal = addp ( sp, PF_1 );
                break;
            case '2':
                nVRetVal = addp ( sp, PF_2 );
                break;
            case '3':
                nVRetVal = addp ( sp, PF_3 );
                break;
            case '4':
                nVRetVal = addp ( sp, PF_4 );
                break;
            case '5':
                nVRetVal = addp ( sp, PF_5 );
                break;
            case '6':
                nVRetVal = addp ( sp, PF_6 );
                break;
            case '7':
                nVRetVal = addp ( sp, PF_7 );
                break;
            case '8':
                nVRetVal = addp ( sp, PF_8 );
                break;
            case '9':
                nVRetVal = addp ( sp, PF_9 );
                break;
            case '-':
            break;
        } // switch
    } // while
} // speak_as_digits

nVRetVal = addp ( sp, PF_MINUS );
break;
case ',' :
    nVRetVal = addp ( sp, PFPOINT );
    break;
default:
    nVRetVal = -1;
    Debug ( 1, sprintf ( Debug_buf,
        "Compute/speak_as_digits: invalid character \d \c' (0x\02X)",
        ( int ) ( chrptr - numbuf ), *chrptr, *chrptr ) );
    break;
}
if ( nVRetVal != RC_SUCCEED )
goto done;
chrptr++;
}
done:
    return ( nVRetVal );
} /* function speak_as_digits */

/****************************************************************************

speak_as_day: Take the input, and convert it to integer. If the integer is in the range 1..7, speak the corresponding day of the week (where Sunday=1). Otherwise, return an error.

returns: 0 = success
-1 = improper day number

***************************************************************************/

static int speak_as_day( const struct line_t *sp,
    const char *numbuf)
{
    int dayval;
    int nVRetVal; // Voysys return value

dayval = atoi ( numbuf );
if ( ( ( dayval < 1 ) || ( dayval > 7 ) ) { nVRetVal = -1; // need a error code ? Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_day: value for day must be between 1 and 7" ) );
goto done;
} } nVRetVal = addp ( sp, Days[dayval - 1] );
done:
    return ( nVRetVal );
} /* function speak_as_day */

***************************************************************************/
speak_0_to_999: Speak numbers in the range of 0-999. This is a utility routine for "speak_as_number". Note that this routine takes an integer as an argument, not a character string.

static int speak_0_to_999 ( 
    const struct line_t *sp, // ptr to slot in line table
    const int intval
) 
{
    int nRetVal;          // Vosys return value
    int hundreds, tens, ones;
    char chrbuf[MAX_CHRBUF];

    /*
     * Break the number into hundreds, tens, and ones.
     */
    hundreds = intval / 100;
    tens = ( intval - ( hundreds * 100 ) ) / 10;
    ones = intval % 10;

    /*
     * Speak the hundreds if they are greater than zero.
     */
    if ( hundreds > 0 ) {
        sprintf ( chrbuf, "%d", hundreds );
        nRetVal = addp ( sp, chrbuf );
        if ( nRetVal != RC_SUCCEEDED )
            goto done;

        nRetVal = addp ( sp, PP_HUNDRED );
        if ( nRetVal != RC_SUCCEEDED )
            goto done;
    }

    /*
     * If the tens digit is one, speak the appropriate "teens"
     * prompt.
     */
    if ( tens == 1 ) {
        sprintf ( chrbuf, "%d", ones );
        nRetVal = addp ( sp, chrbuf );
        if ( nRetVal != RC_SUCCEEDED )
            goto done;
    }

    /*
     * Otherwise, look for both a tens prompt and a ones prompt.
     */
    else {

        /*
         * If the tens digit is greater than 1, speak the
         * appropriate tens prompt.
         */
        if ( tens > 1 ) {
            sprintf ( chrbuf, "%d", tens );
            nRetVal = addp ( sp, chrbuf );
            if ( nRetVal != RC_SUCCEEDED )
                goto done;
        }
/*
 * Then speak the ones prompt, even if it is zero but only
 * if tens and hundreds are zero.
 */
if ((ones != 0) || ((tens == 0) && (hundreds == 0))){
    sprintf(chrbuf, "%d", ones);
    nVRetVal = addp(sp, chrbuf);
    if (nVRetVal != RC_SUCCEEDED)
        goto done;
}
}
done:
return(nVRetVal);
} /* function speak_0_to_999 */

/*******************************************************************************/

speak_as_number: Take in a character string, and speak it as a
number. If it has a decimal point, speak that and speak
the numbers after it as digits. This procedure can handle
the range:
    -999999999999 to 999999999999 (999 billion)

This procedure will ignore all extraneous characters
before the first number (or minus or point).

Returns: 0 = success
         -1 = no numeric characters
         -2 = number is too large

*******************************************************************************/

static int speak_as_number(
    const struct line_t *sp,  // ptr to slot in line table
    const char *nnumbuf)
{
    int numlen, count;
    char chrbuf[MAX_CHRBUF];
    int nVRetVal;  // Voysys return value
    char *chrptr;

    /*
     * Find the first character that is a minus, point, or 0-9.
     * Return an error if there weren't any.
     */
    chrptr = strpbrk(nnumbuf, "0123456789-.");
    if (chrptr == (char *)NULL){
        nVRetVal = -1;  // may need an error code
        Debug(1, sprintf(Debug_buf, "Compute/speak_as_number:
            cannot find any valid character" ));
        goto done;
    }

    /*
     * If the first character is a minus, add that prompt to the
     * list, and move on.
     */
if (*chrptr == '-') {
    nVRetVal = addp ( sp, PF_MINUS );
    if ( nVRetVal != RC_SUCCEEDED )
        goto done;
    chrptr++;
}

/*
 * Find the number of consecutive numeric characters. If the
 * number of digits is greater than twelve (999 billion)
 * return an error.
 */
numlen = strlncpy ( chrptr, "0123456789" );

if ( ((numlen > 12) || (numlen > MAX_CHRBUF )) ) {
    nVRetVal = -2; // may need an error code
    Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_number:
                   number is too large" ) ) ;
    goto done;
}

/*
 * If the number of digits is greater than 9, we have a
 * billions value. Break off the billions digits and
 * speak them, as long as they aren't all zeros.
 */
if ( (numlen > 9) ) {
    count = numlen - 9;
    chrbuf[count] = (char) NULL;
    strncpy ( chrbuf, chrptr, count );

    if ( (atoi ( chrbuf ) > 0 ) ) {
        nVRetVal = speak_0_to_999 ( sp, atoi ( chrbuf ) );
        if ( nVRetVal != RC_SUCCEEDED )
            goto done;

        nVRetVal = addp ( sp, PF_BILLION );
        if ( nVRetVal != RC_SUCCEEDED )
            goto done;
    }
    chrptr += count;
    numlen = 9;
}

/*
 * If the number of digits remaining is greater than 6, we
 * have a millions value. Break off the millions digits
 * and speak them, as long as they aren't all zeros.
 */
if ( (numlen > 6) ) {
    count = numlen - 6;
    chrbuf[count] = (char) NULL;
    strncpy ( chrbuf, chrptr, count );

    if ( (atoi ( chrbuf ) > 0 ) ) {
        nVRetVal = speak_0_to_999 ( sp, atoi ( chrbuf ) );
        if ( nVRetVal != RC_SUCCEEDED )
            goto done;

        nVRetVal = addp ( sp, PF_MILLION );
        if ( nVRetVal != RC_SUCCEEDED )

goto done;
    }
    chrptr += count;
    numlen = 6;
}
/*
 * If the number of digits remaining is greater than 3, we
 * have a thousands value. Break off the thousands
 * digits and speak them, as long as they aren't all
 * zeros.
 */
if ( numlen > 3 ) {
    count = numlen - 3;
    chrbuf[count] = (char) NULL;
    strcpy ( chrbuf, chrptr, count );
    if ( atoi ( chrbuf ) > 0 ) {
        nVRetVal = speak_0_to_999 ( sp, atoi ( chrbuf ) );
        if ( nVRetVal != RC_SUCCEED )
            goto done;
        nVRetVal = addp ( sp, PF_THOUSAND );
        if ( nVRetVal != RC_SUCCEED )
            goto done;
    }
    chrptr += count;
    numlen = 3;
}
/*
 * Now speak the remaining digits. If the value is zero, and
 * there were thousands+ digits, do not speak the zero.
 * If the value is zero, and there were no thousands
 * digits, do speak the zero.
 */
chrbuf[numlen] = (char) NULL;
strcpy ( chrbuf, chrptr, numlen );
    if ( ( atoi ( chrbuf ) > 0 ) || ( numlen < 3 ) ) {
        nVRetVal = speak_0_to_999 ( sp, atoi ( chrbuf ) );
        if ( nVRetVal != RC_SUCCEED )
            goto done;
    }
chrptr += numlen;
/*
 * If the next character is a point (',') , speak it and the
 * next digits. Speak the digits after the decimal point
 * as digits (10.23 = "ten point two three").
 */
if ( *chrptr == '.' ) {
    nVRetVal = addp ( sp, PF_POINT );
    if ( nVRetVal != RC_SUCCEED )
        goto done;
    chrptr++;
    nVRetVal = speak_as_digits ( sp, chrptr );
    if ( nVRetVal != RC_SUCCEED )
        goto done;
}
done:
  return ( nVRetVal );
}
/* function speak_as_number */

/********************************************************************/
speak_as_dollars: Speak a numeric amount as dollars and cents
(or dollars only, if the cents value is zero.) This
procedure can handle the range:
  -999999999 to 999999999 (999 million)
This procedure will ignore all extraneous characters
before the first number (or minus or point).

Returns:  0 = success
          -1 = no numeric characters
          -2 = number is too large

********************************************************************/

static int speak_as_dollars ( const struct line_t *sp,      // ptr to slot in line table
const char *nnumbuf )
{
  char *decptr;
  char *chrptr;
  long dollars;
  int cents;
  char chrbuf[MAXCHRBUF];
  int nVRetVal;      // Voysys return value
  char *nums = "0123456789-.";

  /*
   * Find the first character that is a minus, point, or 0-9.
   * Return an error if there weren’t any.
   */
  chrptr = strpbrk ( nnumbuf, nums );
  if ( chrptr == ( char * ) NULL ) {
    nVRetVal = -1;
    Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_number:
                    cannot find any valid character" ) ) ;
    goto done;
  }

  /* Find the decimal point, if any. If there is one, stick a
   * null there.
   */
  decptr = strchr ( chrptr, ‘.’ );
  if ( decptr ) {
    *decptr = (char) NULL;
  }

  /* If there are any digits before the decimal point, send them
   * off to ‘speak_as_number’. If the return code is
   * non-zero, exit with the same return code.
if ( chrptr != decptr ) {
    nVRetVal = speak_as_number ( sp, chrptr );
    if ( nVRetVal != RC_SUCCEEDED )
        goto done;
    dollars = atol ( chrptr );
}
else
    dollars = 0;

/*
 * Now find out if there is a 'cents' value to speak.  We want
 * to speak a 'cents' if there is a decimal point, and if
 * the two digits after it are non-zero.  Remember to
 * restore the decimal point to the buffer, so as not to
 * modify it.
 */
cents = 0;
if ( decptr ) {
    *decptr = '.';
    decptr++;
    chrbuf[2] = (char) NULL;
    strncpy ( chrbuf, decptr, 2 );
    cents = atoi ( chrbuf );
}

/*
 * If there is a cents value, we now want to speak the prompt
 * "dollars and" (or "dollar and").  Otherwise, just speak
 * "dollars" (or "dollar").  If there is a cents value,
 * speak it, and speak the "cents" (or "cent") prompt.
 */
if ( cents > 0 ) {
    if ( dollars > 0 ) {
        if ( dollars == 1 )
            nVRetVal = addp ( sp, PF_DOLLARAND );
        else
            nVRetVal = addp ( sp, PF_DOLLARSAND );
        if ( nVRetVal != RC_SUCCEEDED )
            goto done;
    }
    nVRetVal = speak_0_to_999 ( sp, cents );
    if ( nVRetVal != RC_SUCCEEDED )
        goto done;
    if ( cents == 1 )
        nVRetVal = addp ( sp, PF_CENT );
    else
        nVRetVal = addp ( sp, PF_CENTS );
    if ( nVRetVal != RC_SUCCEEDED )
        goto done;
}
else if ( dollars > 0 ) {
    if ( dollars == 1 )
        nVRetVal = addp ( sp, PF_DOLLAR );
    else
        nVRetVal = addp ( sp, PF_DOLLARS );
    if ( nVRetVal != RC_SUCCEEDED )
        goto done;
}
done:
    return ( nVRetVal );
}

} /* function speak_as_dollars */

/*--------------------------------------------------------------------------------*/

speak_as_date: Speak a date. Assume the date is in the format
    MM/DD/YY (lower cases not required). Speak the date in
the format specified by the current setting of
    nDateSpeakType:

    MMDDYY: "July 14th, 1980" (month name, ordinal number,
            19XX; only 20th Century dates)

    MMDD:  "July 14th"

    DDMIYY: "the 14th of July, 1980"

    DDMYY:  "the 14th of July"

Returns:
   0    success
   -1   improper format for a date (month)
   -2   improper format for a date (day)
   -3   improper format for a date (year)
   -4   improper format (cannot find '/')

Note: only MMDDYY and MMDD is supported for now !!!!!

/*--------------------------------------------------------------------------------*/

static int speak_as_date (  
    const struct lLine_t *sp,    // ptr to slot in line table
    const char    *nnumbuf
)
{
    int      month, day, year;
    char    *startptr;
    int      nVRetVal;       // Voysys return value

    /*
 *  Extract the month, day, and year from the date. Return -1
 *  if anything at all goes wrong. The month must be at
 *  the beginning of the date.
 */

    startptr = ( char * ) nnumbuf;
    month = atoi ( startptr );
    if ( ( month < 1 ) || ( month > 12 ) ) {
        nVRetVal = -1;
        Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_date: invalid month %d", month ) );
        goto done;
    }

    /*
 *  The day should start after the first slash
 */

    startptr = strchr ( startptr, '/');
    if ( startptr == ( char * ) NULL ) {
        nVRetVal = -4;
    }

}
Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_date: cannot find */*" ) );

goto done;
}

startptr++;  
day = atoi ( startptr );
if ( ( day < 1 ) || ( day > 31 ) ) {
    nRetVal = -2;
    Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_date: invalid day %d", day ) );

goto done;
}

/**<  
* The year should start after the next slash.  
*  
* &&& Note that dBase date is the same as DOS format, which  
* means no provision is made for dates not matching  
* 19**. When they get around to fixing this, we will  
* too.  
*/

startptr = strchr ( startptr, '/' );
if ( startptr == ( char * ) NULL ) {
    nRetVal = -4;
    Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_date: cannot find */*" ) );

goto done;
}

startptr++;
year = atoi ( startptr );
if ( ( year < 0 ) || ( year > 99 ) ) {
    nRetVal = -3;
    Debug ( 1, sprintf ( Debug_buf, "Compute/speak_as_date: invalid year %d", year ) );

goto done;
}

/**<  
* Now add prompts to the prompt list. Start by adding the  
* month and ordinal day prompts, in the order specified  
* by nDateSpeakType.  
*/

if ( ( sp-&gt;nDateSpeakType == DS_MMDDYY ) ||  
     ( sp-&gt;nDateSpeakType == DS_MMDYY ) ) {
    nRetVal = addp ( sp, MonthS[month - 1] );
    if ( nRetVal != RC_SUCCEED )
        goto done;
}

nRetVal = addp ( sp, Ordinals[day - 1] );
if ( nRetVal != RC_SUCCEED )
    goto done;

else {
    nRetVal = addp ( sp, PF_THE );
    if ( nRetVal != RC_SUCCEED )
        goto done;
}

nRetVal = addp ( sp, Ordinals[day - 1] );
if ( nRetVal != RC_SUCCEED )
    goto done;
nRetVal = addp ( sp, PF_OP );
if ( nRetVal != RC_SUCCEEDED )
goto done;

nRetVal = addp ( sp, Months[month - 1] );
if ( nRetVal != RC_SUCCEEDED )
goto done;
}

/*
 * Now add the year prompts, if the nDateSpeakType specifies it. Start with "nineteen", and then speak the year
 * (XX) number as a number - three cases: 00 - speak the prompt "hundred" only 0X - speak the prompt "zer0"
 * then speak the number. XX - simply speak the number.
 * If the year is in the range 1994..2000 and 1990, we have a special prompt for it.
 */

// Note: year 2000 not currently supported.
if ( ( sp->nDateSpeakType == DS-MMDDYY ) || ( sp->nDateSpeakType == DS-DDMMYY ) ) {
    if ( year == 90 )
        nRetVal = addp ( sp, PF_1990 );
    else if ( year == 94 )
        nRetVal = addp ( sp, PF_1994 );
    else if ( year == 95 )
        nRetVal = addp ( sp, PF_1995 );
    else if ( year == 96 )
        nRetVal = addp ( sp, PF_1996 );
    else if ( year == 97 )
        nRetVal = addp ( sp, PF_1997 );
    else if ( year == 98 )
        nRetVal = addp ( sp, PF_1998 );
    else if ( year == 99 )
        nRetVal = addp ( sp, PF_1999 );
    else {
        nRetVal = addp ( sp, PF_19 );
        if ( nRetVal != RC_SUCCEEDED )
            goto done;

        if ( year == 0 ){
            nRetVal = addp ( sp, PF_HUNDRED );
            if ( nRetVal != RC_SUCCEEDED )
                goto done;
        }
    }
    else {
        if ( year < 10 ){
            nRetVal = addp ( sp, PF_0 );
            if ( nRetVal != RC_SUCCEEDED )
                goto done;
        }
        nRetVal = speak_0_to_999 ( sp, year );
        if ( nRetVal != RC_SUCCEEDED )
            goto done;
    }
}

if ( nRetVal != RC_SUCCEEDED )
goto done;

done:
return ( nVRetVal );
}
/* function speak_as_date */

/*****************************/
static void init_play_list (  
    struct line_t *sp          // ptr to slot in line table  
)
{
    struct command cmd;        // command to send  
    struct event  event;      // event received  
    int          nVRetVal;     // Voy sys return value

cmd.nCommand = DVRC_INITPLAY;
sp->nLine = sp->nLineNum;
sp->play_item.type = PT_NONE;
sp->nNumItemsInPlayList = 0;

nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
}
/* function init_play_list */

/*****************************/
static int add_to_play_list (  
    const struct line_t *sp,  // ptr to slot in line table  
    const char *voice_file,  
    const int    play_type
)
{
    int          nVRetVal;
    int          i, vfile_len;
    char         non_digit_char = ( char ) NULL;

    /*
     * Find the first non-digit, non-blank symbol in the string.
     *  If there isn’t any, speak it as a number. If there is
     *  one, and it’s a slash (/), speak the string as a date.
     *  Otherwise, treat the string as a file name.
     */
    vfile_len = strlen ( voice_file );
    for ( i = 0; i < vfile_len; i++ ) {
        if ( ( voice_file[i] != ' ' ) || ( voice_file[i] > '9' ) )  
            if ( ( voice_file[i] != ' ' ) && ( voice_file[i] != ' ' ) )  
                if ( voice_file[i] != '.' )  
                    if ( voice_file[i] != '.' )  
                        if ( voice_file[i] != '.' )  
                            break;
        if ( non_digit_char == ( char ) NULL )
            /* voice file contains a number value */
            switch ( play_type )
            case AS_NUMBER:
                nVRetVal = speak_as_number ( sp, voice_file );
                break;
    }
break;
case AS_DIGITS:
nRetVal = speak_as_digits ( sp, voice_file );
break;
case AS_DOLLARS:
nRetVal = speak_as_dollars ( sp, voice_file );
break;
case AS_DAY:
nRetVal = speak_as_day ( sp, voice_file );
break;
default:
nRetVal = -2;
break;
}
else if ( non_digit_char == '/' ) {
    nRetVal = speak_as_date ( sp, voice_file );
}
else {
    /* voice file contains the name of the voice file to play */
    nRetVal = addu ( sp, voice_file );
}
return ( nRetVal );
} /* function add_to_play_list */

/**************************************************************************/

static int play_play_list (  
    const struct line_t *sp,  // ptr to slot in line table
    const boolean interrupt_mode  // end play if DTMF received?
) {
    struct command  cmd;  // command to send
    struct event  event;  // event received
    int  nRetVal = RC_INTERNAL;  // Voysys return value

    cmd.nCommand = DVRC_PLAY;
    cmd.nline = sp->nLineNum;
    cmd.play_item.type = PI_PHONE;
    cmd.flush_digits_at_start = TRUE;
    cmd.end_voice_on_any_digit = interrupt_mode;
    cmd.end_voice_on_digit_end = FALSE;
    cmd.end_opon_any_digit = FALSE;
    cmd.end_opon_digit_end = TRUE;
    cmd.end_opon_voice_end = ( !interruption_mode );
    cmd.max_digits = 0;
    cmd.end_digit = '#';
    cmd.get_digits_at_end = FALSE;
    cmd.start_timeout = 0;
    cmd.total_timeout = 0;
    cmd.interdigit_to = 0;
    cmd.end_silence = ( unsigned ) NOT_APPLIC;
    cmd.total_recording = ( unsigned ) NOT_APPLIC;

    nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
return ( nRetVal );
} /* function play_play_list */
int com_Initialize(
    const unsigned int hTask, // task handle (Windows only)
    const unsigned int hInstance, // instance handle
    const unsigned long hWnd // hidden window handle (Windows only)
)
{
    int nRetVal; // Vosys return value
    int nSlot; // slot in line table
    struct line_t *sp; // ptr to slot in line table

    nRetVal = CheckDebugSanity();
    if (nRetVal != RC_SUCCEEDED) {
        Debug ( 1, sprintf ( Debug_buf, "Compute/com_Initialize:
            debug sanity failure" ) );
        goto done;
    }

    // find slot in the line table
    nRetVal = FindSlot( hTask, &sp );
    if ( nRetVal == RC_SUCCEEDED ) {
        nRetVal = RC_ALREADYINITIALIZED;
        goto done;
    }

    // find a free slot in the line table
    for ( nSlot = 0 ; nSlot < MAX_LINES ; nSlot++ ) {
        if (!line[nSlot].bInUse)
            break;
    }
    if ( nSlot == MAX_LINES ) {
        Debug ( 5, sprintf ( Debug_buf, "Compute/com_Initialize:
            line table is full" ) );
        nRetVal = RC_SWLIMIT;
        goto done;
    }

    sp = &line[nSlot];
    sp->bInUse = TRUE;
    sp->hTask = hTask;
    sp->bConnected = FALSE;
    sp->nLineNum = MAX_LINES;
    sp->bOnline = FALSE;

    /* initialize lower layer(s) */
    nRetVal = ipc_Initialize( sp->hTask, hInstance, hWnd );
    if ( nRetVal != RC_SUCCEEDED ) {
        sp->bInUse = FALSE;
        goto done;
    }

    /* Set defaults for the various functions.
     */
    sp->nDateSpeakType = DS_NMDDYY;
    sp->lGetWordsStartTimeout = DEF_GETWORDS_START_TIMEOUT;
    sp->lGetTonesStartTimeout = DEF_GETTONES_START_TIMEOUT;
nVRetVal = RC_SUCCEED;

done:
    return ( nVRetVal );
}

/*************************************************************************
* arguments have following meaning:
* - if rline_num is < MAX_LINES, means try for that line only.
* - if rline_num == MAX_LINES, means try for any line.
* - if oline_num is < MAX_LINES, means got that line.
* - if oline_num == MAX_LINES, means did not get a line.
***************************************************************************/

int com_Connect (const unsigned int hTask, // task handle (Windows only)
    const int rline_num, // requested line number
    int *oline_num) // obtained line number
{
    nVRetVal = RC_INTERNAL; // Voysys return value
    struct command cmd;     // command to send
    struct event event;     // event received
    int currencenum;        // line number
    struct line_t *sp;      // ptr to slot in line table
    *oline_num = MAX_LINES;

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCEED)
        goto done;

    if ( (sp->bConnected) ) {
        /* we are already connected to a line; disconnect */
        cmd.nCommand = DVRC_DISCONNECT;
        cmd.nLine = sp->nLineNum;
        cmd.play_item.type = PI_NONE;
        nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

        if ( nVRetVal >= RC_SUCCEED )
            sp->bConnected = FALSE;
        else
            goto done;
    }

    if ( (rline_num < MAX_LINES) ) {
        /* try just one line */
        /* connect to a line */
        cmd.nCommand = DVRC_CONNECT;
        cmd.nLine = rline_num;
        cmd.play_item.type = PI_NONE;
        nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
    }
if ( nVRetVal >= RC_SUCCEED ) {
    sp->bConnected = TRUE;
    sp->nLineNum = rline_num;
    *oline_num = sp->nLineNum;
} else {
    *oline_num = MAX_LINES;
}

else { /* find first free line */
    for (curlinenum = 0; curlinenum < MAX_LINES; curlinenum++){
        /* connect to a line */
        cmd.nCommand = DVRC_CONNECT;
        cmd.nLine = curlinenum;
        cmd.play_item.type = PI_NONE;

        nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

        if ( nVRetVal >= RC_SUCCEED ) {
            sp->bConnected = TRUE;
            sp->nLineNum = curlinenum;
            *oline_num = sp->nLineNum;
            goto done;
        }
    }
    /* failed to find a line */
    *oline_num = MAX_LINES;
}

done:
    return ( nVRetVal );
} /* function com_Connect */

/******************************************************************************
* 
* Tricky: if dir_name == ".\", this means client current directory, not server 
* directory. 
* 
******************************************************************************/
int com_Set_Directory ( const unsigned int hTask, char *dir_type_str, char *dir_name )
{
    int nVRetVal = RC_INTERNAL; // Vosys return value
    struct command cmd;
    struct event event;
    struct line_t *sp;
    int dir_type;

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCEED)
        goto done;

    /* tricky: must be connected to a line before can do a SETDIR */
    if ( !sp->bConnected ) {
        /*
nRetVal = RC_NOTCONNECTED;
goto done;
}

if ( strlen ( dir_name ) > MAX_DN ) {
    nRetVal = RC_BADFILENAME;
goto done;
}

/*
 * Now capitalize the string.
*/
strupr ( dir_type_str );

/*
 * Check through the list of allowable file types. Return an
 * error if the user specified an illegal file type.
*/
if ( !strcmp ( dir_type_str, "SYS_PROMPTS" ) || !strcmp ( dir_type_str, "SYS_PROMPTS" ) ) {
    dir_type = DIR_SYS_PROMPTS;
} else if ( !strcmp ( dir_type_str, "APP_PROMPTS" ) || !strcmp ( dir_type_str, "APP_PROMPTS" ) ) {
    dir_type = DIR_APP_PROMPTS;
} else if ( !strcmp ( dir_type_str, "RECORDINGS" ) ) {
    dir_type = DIR_USERVOICE;
} else if ( !strcmp ( dir_type_str, "FAXFILES" ) ) {
    dir_type = DIR_APP_FAXFILES;
} else if ( !strcmp ( dir_type_str, "TMPFILES" ) ) {
    dir_type = DIR_TMPFILES;
} else {
    nRetVal = RC_SETDIRINVALIDFILETYPE;
goto done;
}

strupr ( dir_name );

/* convert directory name to full path name */
dname_partial_to_full ( dir_name, cmd.name );

/* better late than never ... */
if ( strlen ( cmd.name ) > MAX_DN ) {
    nRetVal = RC_BADFILENAME;
goto done;
}

cmd.nCommand = DVRC_SETDIR;
    cmd.nLine = sp->nLineNum;
    cmd.dir_num = dir_type;
    cmd.play_item.type = PI_NONE;
    nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
}

done:
    return ( nRetVal );

} /* function com_Set_Directory */
int com_Set_Input (const unsigned int hTask, // task handle (Windows only)
const int dev_num // device number)
{
    int nVRetVal = RC_INTERNAL; // Voyysys return value
    struct command cmd; // command to send
    struct event event; // event received
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCESS)
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    if ( sp->bOnLine ) {
        nVRetVal = RC_OFFLINEONLY;
        goto done;
    }

    cmd.nCommand = DAVR_SET_INPUT;
    cmd.nLine = sp->nLineNum;
    cmd.io_location = dev_num;
    cmd.play_item.type = PI_NONE;

    nVRetVal = ipc_Do_Command( sp->hTask, cmd, &event );

done:
    return ( nVRetVal );
} /* function com_Set_Input */

int com_Set_Output (const unsigned int hTask, // task handle (Windows only)
const int dev_num // device number)
{
// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if (nVRetVal != RC_SUCCEEDED)
goto done;
5
if ( !sp->bConnected ) {
nVRetVal = RC_NOTCONNECTED;
goto done;
}
10
if ( sp->bOnLine ) {
nVRetVal = RC_OFFLINEONLY;
goto done;
}
15
cmd.nCommand = DVRC_SETOUTPUT;
cmd.nLine = sp->nLineNum;
cmd.io_location = dev_num;
cmd.play_item.type = PI_NONE;
20
nVRetVal = ipc_Do_Command( sp->hTask, cmd, &event);
done:
return ( nVRetVal );
25
} /* function com_Set_Output */

30 **************************************************************************
*  com_Set_Date_Speak: Set the date speak type.
* **************************************************************************
35
int com_Set_Date_Speak (const unsigned int hTask, // task handle (Windows only)
char *date_type  // date speak type
)
40 {
int nVRetVal = RC_INTERNAL; // Voysys return value
struct line_t *sp; // ptr to slot in line table
int speak_type;
45
// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if (nVRetVal != RC_SUCCEEDED)
goto done;
50
if ( !sp->bConnected ) {
  nVRetVal = RC_NOTCONNECTED;
goto done;
}
55
/*
 * Now capitalize the string.
 */
strupr ( date_type );
60
/*
 * Check through the list of allowable date speak types.
 *  Return an error if the user specified an illegal type.
 */
/*
if (!strcmp (date_type, "MMDDYY")) {
    speak_type = DS_MMDDYY;
}
else if (!strcmp (date_type, "MMDD")) {
    speak_type = DS_MMDD;
}
else if (!strcmp (date_type, "DDMMYY")) {
    speak_type = DS_DDMMYY;
}
else if (!strcmp (date_type, "DDMM")) {
    speak_type = DS_DDMM;
}
else {
    nVRetVal = RC_SETDSPRANINVALIDARG;
    goto done;
}

/*
 * Assume that the nDateSpeakType sent by the upper level is
 * legitimate.
 */
sp->nDateSpeakType = speak_type;
nVRetVal = RC_SUCCEED;

done:
return (nVRetVal);
} /* function com_Set_Date_Speak */

/***************************************************************************/
*
* com_Set_Timeout: set various timeouts
*
***************************************************************************/
int com_Set_Timeout (
    const unsigned int hTask, // task handle (Windows only)
    char *timeout_type_str, // timeout type
    const int timeout // timeout in seconds
)
{
    int nVRetVal = RC_INTERNAL; // Voysys return value
    struct line_t *sp; // ptr to slot in line table
    int timeout_type;
    ulong dwTimeout;

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCEED)
        goto done;

    if (!sp->bConnected) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    /*
     * Make sure the timeout is valid (positive integer).
     */
if ( timeout <= 0 ) {
    nVRetVal = RC_SETTMOUTBADTIMEOUT;
    goto done;
}

/*
 * Convert the timeout to milliseconds.
 */
dwTimeout = (ulong) timeout * 1000;

/*
 * Now capitalize the string.
 */
strupr ( timeout_type_str );

/*
 * Check through the list of allowable timeout types. Return an error if the user specified an illegal output type.
 */
if ( !strcmp ( timeout_type_str, "VGETTONES" ) ) {
    timeout_type = TO_VGETTONES;
    sp->lGetTonesStartTimeout = dwTimeout;
}
else if ( !strcmp ( timeout_type_str, "VGETWORDS" ) ) {
    timeout_type = TO_VGETWORDS;
    sp->lGetWordsStartTimeout = dwTimeout;
}
else {
    nVRetVal = RC_SETTMOUTBADKEYWORD;
    goto done;
}

nVRetVal = RC_SUCCEED;

done:
return ( nVRetVal );
} /* function com_Set_Timeout */

/**
 * Open an indexed prompt file
 */

int com_Set_Indexed_File ( const unsigned int hTask, // task handle (Windows only)
                          const char *base_name : // base filename (no '.ext' on end)
                          )
{
    int nVRetVal = RC_INTERNAL; // Voysys return value

    int i;
    struct command cmd; // command to send
    struct event *event; // event received
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if ( nVRetVal != RC_SUCCEED)
- 146 -

goto done;

if ( !sp->bConnected ) {
    nRetVal = RC_NOTCONNECTED;
    goto done;
}

if ( strlen ( base_name ) >= MAX_FN - 4 ) {
    nRetVal = RC_BADCOMMAND;
    goto done;
}

cmd.nCommand = DVRC_SETFILE;

5

cmd.nLine = sp->nLineNum;

10

cmd.play_item.type = PI_NONE;

15

strcpy ( cmd.play_item.fname, base_name );

strcpy ( cmd.name, base_name );

strcat ( cmd.name, "_map" );

cmd.play_item.index = -1;

20

cmd.play_item.vdata = { void * } 0xffffffff;

cmd.path[0] = DIR_SYSINPUT;

for ( i = 1; i < MAX_DIRS; i++ )

25

cmd.path[i] = DIR_END;

done:

    return ( nRetVal );

30

} /* function com_Set_Indexed_File */

/**************************************************************************

35

* Tell lower levels what vocabulary we are using for
* voice-recognition. There are 3 relevant data files, all with
* the same base name:
 *
* basename.va  Vocabulary file. Rhetorex proprietary format.
*               Name is specified in driver config file; driver
*               reads in this file.
 *
40
* basename.vmp Vocabulary "word map" file. One word-name per
* line. Line number == word index in vocabulary
* file (first line == 1). File must be in
* SYSPROMPTS directory.
 *
* basename.vsm Vocabulary "syntax map" file. One syntax-name
* per line. Line number == syntax mask value
* (first line == 0). File must be in SYSPROMPTS
* directory.
 *
***************************************************************************/

45

50

55

int com_Set_Vocabulary (  
    const unsigned int hTask, // task handle (Windows only)
    const char *base_name // base filename (no ".ext" on end)
)

60
{
    int nRetVal = RC_INTERNAL; // Voysys return value
    int i;

- 147 -

struct command cmd;  // command to send
struct event event;  // event received
struct line_t *sp;   // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if (nVRetVal != RC_SUCCESS)
goto done;

if ( !sp->bConnected ) {
    nVRetVal = RC_NOTCONNECTED;
goto done;
}

if ( strlen( base_name ) >= MAX_FN - 4 ) {
    nVRetVal = RC_BADCOMMAND;
    goto done;
}

cmd.nCommand = DVRC_SETVOCAB;
cmd.nLine = sp->nLineNum;
cmd.play_item.type = PI_NONE;
strcpy( cmd.play_item.fname, base_name );
strcat( cmd.play_item.fname, ".vmp" );
strcpy( cmd.name, base_name );
strcat( cmd.name, ".vsm" );
cmd.play_item.index = -1;
cmd.play_item.vdata = ( void * ) 0xFFFFFFFF;
cmd.path[0] = DIR_VOCABS;
for ( i = 1; i < MAX_DIRS; i++ )
    cmd.path[i] = DIR_END;

nVRetVal = ipc_Do_Command( sp->hTask, cmd, &event );

if ( !sp->bConnected ) {
    nVRetVal = RC_NOTCONNECTED;
goto done;
}

} /* function com_Set_Vocabulary */

*******************************************************************************/

int com_Disconnect( const unsigned int hTask )  // task handle (Windows only)
{
    int nVRetVal = RC_INTERNAL;  // Vcysys return value
    struct command cmd;  // command to send
    struct event event;  // event received
    struct line_t *sp;   // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCESS)
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
goto done;
    }
}
if ( sp->bOnLine ) {
    nVRetVal = RC_CARDOFFHOOK;
    goto done;
}

cmd.nCommand = DVRC_DISCONNECT;
cmd.nLine = sp->nLineNum;
cmd.play_item.type = PI_NONE;

nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

if ( nVRetVal >= RC_SUCCEED ) {
    sp->bConnected = FALSE;
    sp->nLineNum = MAX_LINES;
    sp->bOnLine = FALSE;
}

done:
    return ( nVRetVal );

} /* function com_Disconnect */

int com_Shutdown ( const unsigned int hTask ) // task handle (Windows only)
{
    struct command cmd; // command to send
    struct event event; // event received
    int nVRetVal = RC_INTERNAL; // Voysys return value
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if ( nVRetVal != RC_SUCCEED )
        goto done;

    if ( sp->bConnected ) {
        if ( sp->bOnLine ) {
            /* should hang up here !!!

            */
            cmd.nCommand = DVRC_DISCONNECT;
            cmd.nLine = sp->nLineNum;
            cmd.play_item.type = PI_NONE;

            nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
            /* ignore return code */
            sp->bConnected = FALSE;
            sp->bOnLine = FALSE;
            sp->nLineNum = MAX_LINES;
        }
    }

    nVRetVal = ipc_Shutdown ( sp->hTask );

    sp->bInUse = FALSE;

    done:
        return ( nVRetVal );
int com_Answer (const unsigned int hTask, // task handle (Windows only)
    const int num_rings, // answer incoming after X rings
    const int total_timeout // seconds
) {
    struct command cmd; // command to send
    struct event event; // event received
    int nVRetVal = RC_INTERNAL; // Voyys return value
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot(hTask, &sp);
    if (nVRetVal != RC_SUCCEEDED)
        goto done;

    if (!sp->bConnected) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    if (sp->bOnLine) {
        nVRetVal = RC_CARDOFFHOOK;
        goto done;
    }

    /*
     * Validate the parameters.
     */
    if (num_rings < 1) {
        nVRetVal = RC_WAITRINGINVALIDNUMRINGS;
        goto done;
    }

    if (total_timeout < 0) {
        nVRetVal = RC_WAITRINGINVALIDTIMEOUT;
        goto done;
    }

    cmd.nCommand = DVRC_WAITFORCALL;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PI_NONE;
    cmd.answer_rings = num_rings;
    cmd.total_timeout = total_timeout * 1000; // convert to milliseconds

    nVRetVal = ipc_Do_Command(sp->hTask, cmd, &event);

    if (nVRetVal == RC_SUCCEEDED)
        sp->bOnLine = TRUE;

    done:
    return (nVRetVal);
}

} /* function com_Answer */
- 150 -

/**************************/

/* Go off-hook, dial number, wait for answer */

5

/*
 * syntax inside outdialnum is:
 * *
 * 0123456789 ABCD         digit to dial
 * T                       use tone dialing (default)
 * P                       use pulse dialing
 * W                       wait for dial tone, pause 2 seconds
 * !                       flash the switch-hook
 */

15

int com_Callout ( 
    const unsigned int hTask, // task handle (Windows only)
    /*
     * syntax inside outdialnum is:
     * *
     * 0123456789 ABCD         digit to dial
     * T                       use tone dialing (default)
     * P                       use pulse dialing
     * W                       wait for dial tone
     * !                       flash the switch-hook
     */
     const char *outdialnum,
    const int num_rings,    // give up after X rings
    const boolean wait_for_silence, // TRUE if predictive
dialing
    ulong *voice_after_answer // noise after answer (msec)
)
{

35

    struct command cmd;       // command to send
    struct event event;       // event received
    int nVRetVal = RC_INTERNAL;   // Voysys return value
    ulong basedialtime;        // time (msec) to dial dialstring

40

    const char *p;
    struct line_t *sp;        // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );

45

    if (nVRetVal != RC_SUCCED)
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

50

    if ( sp->bOnLine ) {
        nVRetVal = RC_CARDOFFHOOK;
        goto done;
    }

55

    if (num_rings < 1) {
        nVRetVal = RC_DIALBADNUMRINGS;
        goto done;
    }
if ( strlen ( outdialnum ) >= MAXDIGITS ) {
    nRetVal = RC_BADCOMMAND;  // Note: need a new error code
    goto done;
}

/* see if there are any illegal characters in outdialnum */
if ( strspn ( outdialnum, "0123456789#*ABCDTPW,!" ) != strlen ( outdialnum ) ) {
    nRetVal = RC_BADCOMMAND;  // Note: need a new error code
    goto done;
}

basetime = BASEDIALTIME_W;
p = outdialnum;
while ( *p != '\0' ) {
    switch ( *p++ ) {
        case 'W':
            basetime += BASEDIALTIME_W;
            break;  // wait for dialtone == 3 secs ?
        case ',':
            basetime += BASEDIALTIME_COMMA;
            break;  // pause == 2 secs
        default:
            basetime += BASEDIALTIME_DEFAULT;
            break;  // digit == 0.2 secs ?
    }
}

cmd.nCommand = DVRC_CALLOUT;
cmd.nLine = sp->nLineNum;
cmd.play_item.type = PI_NONE;
cmd.answer_rings = num_rings;
cmd.wait_for_silence = wait_for_silence;
strcpy (cmd.string, outdialnum);
cmd.total_timeout = basetime + ( 60000L * ( (long) (num_rings + 1) ) );

nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
*voice_after_answer = event.voice_after_answer;
if (nRetVal == RC_SUCCEEDED)
    sp->bOnLine = TRUE;

// convert return code
if (nRetVal == RC_BUSYDEST)
    nRetVal = RC_DIALGOTBUSYDEST;
else if (nRetVal == RC_NOANSWER)
    nRetVal = RC_DIALGOTNOANSWER;

done:
    return ( nRetVal );

} /* function com_Callout */

/*****************************/

/* Just send digits out; no fancy stuff */
int com_Dial {

const unsigned int hTask, // task handle (Windows only)
const char * digits
{
    struct command cmd; // command to send
    struct event event; // event received
    int nRetVal = RC_INTERNAL; // Voyys return value
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nRetVal = FindSlot( hTask, &sp );
    if ( nRetVal != RC_SUCCESS )
        goto done;

    if ( !sp->bConnected ) {
        nRetVal = RC_NOTCONNECTED;
        goto done;
    }

    if ( !sp->bOnLine ) {
        nRetVal = RC_CARONHOOK;
        goto done;
    }

    if ( strlen( digits ) >= MAX_DIGITS ) {
        nRetVal = RC_BADCOMMAND; // Note: need a new error code
        goto done;
    }

    /* see if there are any illegal characters in digits */
    /* note: can not do "W" in this function */
    if ( strspn( digits, "0123456789#*ABCOTP,!:" ) != strlen( digits ) ) {
        nRetVal = RC_BADCOMMAND; // Note: need a new error code
        goto done;
    }

cmd.nCommand = DVRC_SENDDTMF;
cmd.nLine = sp->nLineNum;
cmd.play_item.type = PI_NONE;
strcpy( cmd.string, digits );

    nRetVal = ipc_Do_Command( sp->hTask, cmd, &event );

    done:
    return( nRetVal );
}

/*==========================================================================*/

/*
 * Get DTMF digits entered by user.
 * 
 * Does not flush digits at start, so any digits left around from
 * a previous operation will be seen.
 */
int com_Get_Tones (
    const unsigned int hTask,  // task handle (Windows only)
    const boolean play_playlist, // play playlist?
)

    * Values of term_digits:
    * ==0  unlimited (limited only by line card).
    * >0   limited to term_digits digits.
    */
    const int term_digits, // end op after X keys recv

    * Values of term_key:
    * '?'  any digit terminates operation.
    * ','  no term digit; end by timeout or term_after_x_keys.
    * 'x'  terminate operation if digit 'x' input.
    */
    const char term_key, // end op if this key recv
    const int interdigit_timeout, // seconds
    const int retries,  // if timeout, retry N times
    char *outval  // retval -- digits received

    }

    struct command cmd;  // command to send
    struct event event;  // event received
    int nRetVal = RC_INTERNAL;  // Voysys return value
    int i;
    char *cptr;  // used to strip off terminating key
    struct line_t *sp; // ptr to slot in line table
    ulong term_id_timeout; // interdigit timeout to
    terminate on in msec

    *outval = '\0';

    // find slot in the line table
    nRetVal = FindSlot ( hTask, &sp );

    if ( nRetVal != RC_SUCCESS )
    goto done;

    if ( !sp->bConnected ) {
        nRetVal = RC_NOTCONNECTED;
        goto done;
    }

    if ( !sp->bOnLine ) {
        nRetVal = RC_CARDONHOOK;
        goto done;
    }

    if ( retries < 0 ) {
        nRetVal = RC_BADCOMMAND;
        goto done;
    }

    if ( ( term_digits < 0 ) || ( term_digits > MAX_DIGITS ) ) {
        Debug ( 1, sprintf ( Debug_buf, "Compute/com_Get_Tones:
        bad number of digits for termination" ) );
        nRetVal = RC_GETTONESBADTERMDIGITS;
        goto done;
    }

    if ( !strchr ( "0123456789#$", term_key ) ) {
        Debug ( 1, sprintf ( Debug_buf, "Compute/com_Get_Tones:
        bad termination key" ) );
    }
nRetVal = RC_GETTONESBADTERMKEY;
goto done;
}

if ( interdigit_timeout < 0 ) {
    Debug ( 1, sprintf ( Debug_buf, "Compute/com_Get_Tones:
               bad interdigit timeout value" ) );
    nRetVal = RC_GETTONESBADINTDIGITTIMEOUT;
    goto done;
}

/* Convert the timeouts to milliseconds. */
term_id_timeout = interdigit_timeout * 1000;

for ( i = 0; i < retries + 1; i++ ) {
    if ( play_playlist ) {
        cmd.nCommand = DVRC_PLAY;
        cmd.nLine = sp->nLineNum;
        cmd.play_item.type = PI_NONE;
        cmd.flush_digits_at_start = FALSE;
        cmd.end_voice_on_any_digit = TRUE;
        cmd.end_voice_on_digit_end = FALSE;
        cmd.end_oper_on_any_digit = TRUE;
        cmd.end_oper_on_digit_end = FALSE;
        cmd.end_oper_on_voice_end = TRUE;
        cmd.max_digits = term_digits;
        cmd.end_digit = term_key;
        cmd.get_digits_at_end = FALSE;
        cmd.start_timeout = sp->lGetTonesStartTimeout;
        cmd.total_timeout = 0;
        cmd.interdigit_to = term_id_timeout;
        cmd.end_silence = ( unsigned ) NOT_APPLIC;
        cmd.total_recording = ( unsigned ) NOT_APPLIC;

        nRetVal = ipc_Do_Command ( sp->hTask, cmd, kEvent );

        if ( nRetVal < RC_SUCCESS )
            break;
    }

    cmd.nCommand = DVRC_GETDTMF;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PI_NONE;
    cmd.flush_digits_at_start = FALSE;
    cmd.end_voice_on_any_digit = FALSE;
    cmd.end_voice_on_digit_end = FALSE;
    cmd.end_oper_on_any_digit = ( term_key == '?' );
    cmd.end_oper_on_digit_end = ( ( term_key != '?' ) &&
                                 ( term_key != '' ) );

    cmd.end_oper_on_voice_end = FALSE;
    cmd.max_digits = term_digits;
    cmd.end_digit = term_key;
    cmd.get_digits_at_end = true;
    cmd.start_timeout = sp->lGetTonesStartTimeout;
    cmd.total_timeout = 0;
    cmd.interdigit_to = term_id_timeout;
    cmd.end_silence = ( unsigned ) NOT_APPLIC;
    cmd.total_recording = ( unsigned ) NOT_APPLIC;
nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
if ( nRetVal < RC_SUCCEEDED )
    break;
if ( event.string[0] != '\0' ) {
    /* if string ends with term_key, strip it off */
    /* this also could be done by checking for RC_ENDFLAG */
    cptr = event.string;
    while ( *(cptr++) != '\0' )
        cptr++;
    cptr--;
    if ( *(cptr == term_key )
        *cptr = '\0';
    break;
}
}
if ( nRetVal >= RC_SUCCEEDED )
    strcpy ( outval, event.string );
}
} /* function com_Get_Tones */

/**************************************************************/
/*
 * Get words spoken by the user and voice-recognized.
 */
/*
 * Values of term_after_x_words: ==0 unlimited (limited only by
 * line card). >0 limited to term_after_x_words words.
 */
int com_Get_Words ( const unsigned int hTask, // task handle (Windows only)
        const boolean play_playlist, // play playlist ?
        const char *sub_vocab_name,
        /* Values of term_after_x_words:
        * ==0 unlimited (limited only by line card).
        * >0 limited to term_after_x_words words.
        */
        const int term_after_x_words, // end op after X words
        const ulong interword_timeout, // msec
        const int retries, // if timeout, retry N times
        char *outval, // retval -- words received
    )
{
    struct command cmd; // command to send
    struct event event; // event received
    int nRetVal = RC_INTERNAL; // Vosys return value
    int i;
    struct line_t *sp; // ptr to slot in line table
    *outval = '\0';
    // find slot in the line table
nRetVal = FindSlot(hTask, &sp);
if (nRetVal != RC_SUCCEEDED)
goto done;

if (!sp->bConnected ) {
    nRetVal = RC_NOTCONNECTED;
    goto done;
}

if (!sp->bOnline ) {
    nRetVal = RC_CARDONHOOK;
    goto done;
}

if ( retries < 0 ) {
    nRetVal = RC_BADCOMMAND;
    goto done;
}

for ( i = 0; i < retries + 1; i++ ) {
    if ( play_playlist ) {
        cmd.nCommand = DVRC_PLAY;
        cmd.nLine = sp->nLineNum;
        cmd.play_item.type = PL_NONE;
        cmd.flush_digits_at_start = FALSE;
        cmd.end_voice_on_any_digit = TRUE;
        cmd.end_voice_on_digit_end = FALSE;
        cmd.end_oper_on_digit_end = FALSE;
        cmd.end_oper_on_voice_end = TRUE;
        cmd.max_digits = 1;
        cmd.end_digit = '9';
        cmd.get_digits_at_end = FALSE;
        cmd.start_timeout = 0;
        cmd.total_timeout = 0;
        cmd.interdigit_to = 0;
        cmd.end_silence = ( unsigned ) NOT_APPLIC;
        cmd.total_recording = ( unsigned ) NOT_APPLIC;
        nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
        if ( nRetVal < RC_SUCCEEDED )
            break;
    }

    cmd.nCommand = DVRC_GETWORDS;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PL_NONE;
    cmd.max_digits = term_after_x_words;
    cmd.start_timeout = sp->lGetWordsStartTimeout;
    cmd.total_timeout = 0;
    cmd.interdigit_to = interword_timeout;
    cmd.end_silence = ( unsigned ) NOT_APPLIC;
    cmd.total_recording = ( unsigned ) NOT_APPLIC;
    cmd.max_score = 400;
    cmd.min_ambiguity = 20;
    cmd.input_gain = 0x7000;
    strcpy ( cmd.name, sub_vocab_name );

    nRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
    if ( nRetVal < RC_SUCCEEDED )
- 157 -

goto done;
for ( i = 0; i < MAX_WORDS; i++ ) {
    strcat ( outval, event.words[i] );
    // strcat( outval, " ");  /* separate the words with
    */
}

goto done;
}

done:
    return ( nVRetVal );
}  /* function com_Get_Words */

int com_Hangup ( const unsigned int hTask )  // task handle (Windows only)
{
    struct command cmd;  // command to send
    struct event event;  // event received
    int nVRetVal = RC_INTERNAL;  // Voyvys return value
    struct line_t *sp;  // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );

    if ( nVRetVal != RC_SUCCEED )
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    if ( !sp->bOnLine ) {
        nVRetVal = RC_SUCCEED;
        goto done;
    }

    cmd.nCommand = DVRC_HANGUP;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PI_NONE;

    nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

    if ( nVRetVal != RC_SUCCEED )
        sp->bOnLine = FALSE;

    done:
        return ( nVRetVal );
}

}  /* function com_Hangup */

int com_Hangup_Detect ( const unsigned int hTask )  // task handle (Windows only)
struct command cmd; // command to send
struct event event; // event received
int nVRetVal = RC_INTERNAL; // Voysys return value
struct line_t *sp; // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if (nVRetVal != RC_SUCCEEDED)
goto done;
if (!sp->bConnected)
    nVRetVal = RC_NOTCONNECTED;
goto done;
}
cmd.nCommand = DVRC_GETSTATE;
cmd.nLine = sp->nLineNum;
cmd.play_item.type = PI_NONE;
nVRetVal = ipc_Do_Command( sp->hTask, cmd, &event );
if ( nVRetVal >= RC_SUCCEEDED ) {
    if ( event.user_off_hook )
        nVRetVal = RC_OFFHOOK;
    else
        nVRetVal = RC_ONHOOK;
}
done:
    return ( nVRetVal );
} /* function com_Hangup_Detect */

/**************************************************************************/

int com_Play( 
    const unsigned int hTask, // task handle (Windows only)
    const int call_type,
    const char *voice_file, // one filename, a num, or a date
    const int play_type,
    const int interrupt_mode
){
    int nVRetVal = RC_INTERNAL; // Voysys return value
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCEEDED)
        goto done;
    // Note: to support speak for online and not online
    // should not check for bOnLine here and it will be checked
    // in the line driver layer of the server
    if (!sp->bConnected)
        nVRetVal = RC_NOTCONNECTED;
goto done;
if ((play_type < AS_NUMBER) || (play_type > AS_DAY)) {
    nRetVal = -10;
    goto done;
}

switch (call_type) {
    case IMMEDIATE: // not use for now
        init_play_list (sp);
        //
        if (add_to_play_list (sp, voice_file, play_type) != RC_NOFILE)
            if (add_to_play_list (sp, voice_file, play_type) != RC_SUCCEEDED)
                nRetVal = RC_SPEAKPROMPTFILENOTFOUND;
            else
                nRetVal = play_play_list (sp, interrupt_mode);
        break;

    case START_PLAY_LIST:
        init_play_list (sp);
        if (add_to_play_list (sp, voice_file, play_type) != RC_SUCCEEDED)
            nRetVal = RC_SPEAKPROMPTFILENOTFOUND;
        break;

    case ADD_PLAY_LIST:
        sp->nNumItemsInPlayList++;
        if (add_to_play_list (sp, voice_file, play_type) != RC_SUCCEEDED)
            nRetVal = RC_SPEAKPROMPTFILENOTFOUND -
            sp->nNumItemsInPlayList;
        break;

    /*
    * PLAY_PLAY_LIST no longer adds to the play list -
    * just plays the list
    */
    case PLAY_PLAY_LIST:
        nRetVal = play_play_list (sp, interrupt_mode);
        break;

    default:
        nRetVal = -12;
}

done:
    return (nRetVal);
}

/*============================================================================*/
int com_Record (const unsigned int hTask, // task handle (Windows only)
const char *msgname, // null == generate name;
const boolean play_playlist, // play playlist before record ?
const boolean play_playlist, // play playlist before record ?
const boolean play_playlist, // play playlist before record ?
const boolean play_playlist, // play playlist before record ?
const boolean play_playlist, // play playlist before record ?
/*
 * Values of term_key:
 * '0' any digit terminates operation.
 * '9' no term digit; end by timeout or term_after_x_keys.
 * 'x' terminate operation if digit 'x' input.
 */

const char term_key,  // end op if this key recvd
const ulong silence_timeout,  // msec
const int max_recording // maximum length of the
                        // recording in seconds

struct command cmd;        // command to send
struct event event;       // event received
int nVRetVal = RC_INTERNAL;  // Voysys return value
int i;
struct line_t *sp;       // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if ( nVRetVal == RC_SUCCESS )
goto done;

// Note: to support record for online and not online
// should not check for bOnLine here as it will be checked
// in the line driver layer of the server
if ( !sp->bConnected ) {
    nVRetVal = RC_NOTCONNECTED;
goto done;
}

if ( strlen( msgname ) >= ( MAX_DN + MAX_FN ) ) {
    nVRetVal = RC_RECORDINVALIDFILENAME;
goto done;
}

/*
 * Validate the "max_recording_length" parameter.
 */

if ( max_recording < 1 ) {
    nVRetVal = RC_RECORDINVALIDMAXCLLEN; // need to match
    whatever is defined in
    the ERS
    goto done;
}

if ( play_playlist ) {
    cmd.nCommand = DVRC_PLAY;
    cmd.nLine = sp->nLineNum;
    cmd.play_item.type = PI_NONE;
    cmd.flush_digits_at_start = TRUE;
    cmd.end_voice_on_any_digit = ( term_key == '?' );
    cmd.end_voice_on_digit_end = ( ( term_key != '9' ) &&
    ( term_key != '9' ) );
    cmd.end_oper_on_any_digit = ( term_key == '?' );
    cmd.end_oper_on_digit_end = ( ( term_key != '9' ) &&
    ( term_key != '9' ) );
    cmd.end_oper_on_voice_end = TRUE;
    cmd.max_digits = 8;
    cmd.end_digit = term_key;
    cmd.get_digits_at_end = FALSE;
}
cmd.start_timeout = 0;
cmd.total_timeout = 0;
cmd.interdigit_to = 0;
cmd.end_silence = (unsigned) NOT_APPLIC;
cmd.total_recording = (unsigned) NOT_APPLIC;

5
nRetVal = ipc_Do_Command (sp->hTask, cmd, &event);
if (nRetVal < RC_SUCCEED)
        goto done;
}

10
  cmd.nCommand = DVRC_RECORD;
cmd.nLine = sp->nLineNum;
cmd.path[0] = DIR_USERVOICE;
for (i = 1; i < MAX_DIRS; i++)
        cmd.path[i] = DIR_END;
cmd.play_item.type = PI_SINGLE_FILE;
strcpy (cmd.play_item.fname, msgname);

20
cmd.flush_digits_at_start = TRUE;
cmd.end_voice_on_any_digit = (term_key == '7');
cmd.end_voice_on_digit_end = ((term_key != '7') && (
                              term_key != ' ')));
cmd.end_oper_on_any_digit = (term_key == '7');
cmd.end_oper_on_digit_end = ((term_key != '7') && (
                              term_key != ' '));
cmd.end_oper_on_voice_end = (term_key == ' ');
cmd.max_digits = 8;
cmd.end_digit = term_key;
cmd.get_digits_at_end = FALSE;
cmd.start_timeout = 0;
cmd.total_timeout = 0;
cmd.interdigit_to = 0;
cmd.end_silence = silence_timeout;
cmd.total_recording = max_recording;

25
nRetVal = ipc_Do_Command (sp->hTask, cmd, &event);

30
done:
        msgname[0] = '\0';
if (nRetVal >= RC_SUCCEED)
        strcpy (msgname, event.filename);
return (nRetVal);

35}

40
/* function com_Record */

45
#endif

50
/**
  * @brief
  */

55
int com_Faxcover (  
    const unsigned int hTask, // task handle (Windows only)
    const int cp_type, // cover page options
    const void *lpParm // parameter string
)
{
    struct command cmd; // command to send
    struct event event; // event received
    int nRetVal = RC_INTERNAL; // Voysys return value

struct line_t *sp;         // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCESS)
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    cmd.nCommand = DVRC_FAXCOVER;
    cmd.nLine = sp->nLineNum;
    cmd.nFaxOpCode = op_type;
    cmd.nFaxParm = 0;

    if ( lpParm )
        _fstrncpy( cmd.cFaxParm, lpParm, MAX_FAXPARM );
    else
        cmd.cFaxParm[0] = '\0';

    nVRetVal = ipc_Do_Command( sp->hTask, cmd, &event );

done:
    return (nVRetVal);
}

} /* function com_Faxcover */

/***********************************************************/

int  com_Faxdoc ( const unsigned int hTask, // task handle (Windows only)
    const int op_type,   // document page options
    const void far *lpParm // parameter string
)
{
    struct command cmd;       // command to send
    struct event event;       // event received
    int nVRetVal = RC_INTERNAL; // Voyays return value
    struct line_t *sp;        // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCESS)
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    cmd.nCommand = DVRC_FAXDOC;
    cmd.nLine = sp->nLineNum;
    cmd.nFaxOpCode = op_type;
    cmd.nFaxParm = 0;

    if ( lpParm )
        _fstrncpy( cmd.cFaxParm, lpParm, MAX_FAXPARM );
    else
cmd.cFaxParm[0] = '\0';

nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );
}

return ( nVRetVal );
} /* function com_Faxdoc */

int com_Faxsend (
const unsigned int hTask,       // task handle (Windows only)
const char far *lpDia1Str,      // the destination fax number
const int op_type,              // document page options
const void far *lpParm          // parameter string
)
{
struct command cmd;              // command to send
struct event event;             // event received
int nVRetVal = RC_INTERNAL;      // Voysys return value
struct line_t *sp;              // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if (nVRetVal != RC_SUCCEED)
goto done;

if ( !sp->bConnected ) {
    nVRetVal = RC_NOTCONNECTED;
goto done;
}

    cmd.nCommand = DVRC_FAXSEND;
    cmd.nLine = sp->nLineNum;
    cmd.nFaxOpCode = op_type;
    cmd.nFaxParm = 0;
    _fstrncpy( cmd.string, lpDia1Str, MAX_DIGITS );
    if ( lpParm )
        _fstrncpy( cmd.cFaxParm, lpParm, MAX_FAXPARAM );
    else
        cmd.cFaxParm[0] = '\0';

nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

done:
    return ( nVRetVal );
} /* function com_Faxsend */

int com_Faxsetup (  
const unsigned int hTask,       // task handle (Windows only)
const int op_type,              // fax setup options
const void far *lpParm          // string parameter
const int nParm                  // integer parameter
)


```c
struct command  cmd;  // command to send
struct event    event;  // event received
int             nVRetVal = RC_INTERNAL;  // VoySys return value
struct line_t   *sp;  // ptr to slot in line table

// find slot in the line table
nVRetVal = FindSlot( hTask, &sp );
if (nVRetVal != RC_SUCCEED)
goto done;

if (!sp->bConnected) {
    nVRetVal = RC_NOTCONNECTED;
goto done;
}

cmd.nCommand = DVRC_FAXSETUP;
const int nLine = sp->nLineNum;
const int nFaxOpCode = op_type;
const int nFaxParm = nParm;
if ( lpParm )
    strncpy( cmd.cFaxParm, lpParm, MAX_FAXPARM );
else
    cmd.cFaxParm[0] = '\0';

nVRetVal = ipc_Do_Command( sp->hTask, cmd, &event );
done:
    return ( nVRetVal );
}

```

```c
int com_Set (const unsigned int hTask, // task handle (Windows only)
               const int  op_type, // set options
               const void far *lpParm, // string parameter
               const ulong   dwParm  // unsigned long parameter
               )
{
    struct command  cmd;  // command to send
    struct event    event;  // event received
    int             nVRetVal = RC_INTERNAL;  // VoySys return value
    struct line_t   *sp;  // ptr to slot in line table

    // find slot in the line table
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCEED)
goto done;

    if (!sp->bConnected) {
        nVRetVal = RC_NOTCONNECTED;
goto done;
    }

    cmd.nCommand = DVRC_SET;
    cmd.nLine = sp->nLineNum;
}
```
cmd.nSetGetOpCode = op_type;
cmd.dwSetParm = dwParm;
if ( lpParm )
    _fstrncpy( cmd.cSetParm, lpParm );
else
    cmd.cSetParm[0] = ( char ) NULL;

nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

done:
    return ( nVRetVal );
}

/* function com_Set */

/******************************/

int com_Get (  
    const unsigned int hTask,  // task handle (Windows only)  
    const int op_type, // get options  
    void far *lpParm, // string parameter  
    ulong *dwParm // unsigned long parameter  
    )  
{
    struct command cmd; // command to send  
    struct event event; // event received  
    int nVRetVal = RC_INTERNAL; // Voysys return value  
    struct line_t *sp; // ptr to slot in line table  

    // find slot in the line table  
    nVRetVal = FindSlot( hTask, &sp );
    if (nVRetVal != RC_SUCCEEDED)
        goto done;

    if ( !sp->bConnected ) {
        nVRetVal = RC_NOTCONNECTED;
        goto done;
    }

    cmd.nCommand = DVRC_GET;
    cmd.nLine = sp->nLineNum;
    cmd.nSetGetOpCode = op_type;
    cmd.dwSetParm = *dwParm;
    cmd.cSetParm[0] = ( char ) NULL;

    switch ( op_type ) {
    case GET_RECORDINGS:
        cmd.dir_num = DIR_USERVOICE;
        break;

    case GET_TMPFILES:
        cmd.dir_num = DIR_TMPFILES;
        break;

    }  

    nVRetVal = ipc_Do_Command ( sp->hTask, cmd, &event );

    if ( nVRetVal >= RC_SUCCEEDED ) {
        *dwParm = event.dwGetParm;
        if ( event.cGetParm[0] != ( char ) NULL )
            _fstrncpy( lpParm, event.cGetParm );
    }

}
else
    lpParm = ( char * ) NULL;
}

done:
return ( nVRetVal );
} /* function com_Get */

FUNCTION: comProcessEvent

PURPOSE: Handles all messages received by the client application.

int comProcessEvent ( const unsigned int hTask, // task handle (Windows only)
            const struct event event // event from above
)
{
    int nVRetVal; // Voysys return value

    // don't even try to find table slot!
    nVRetVal = ipcProcessEvent ( hTask, event );
    return ( nVRetVal );
} /* function comProcessEvent */
APPENDIX E
IPC_WD_C.c
Copyright 1992-1994 Vosys Corporation

5    /********************************************************************************
   * File: IPC_WD_C.c
   * Top part of InterProcess Communication layer for multiple-line
   * system. Independent of line card type.
   * Bottom layer of Client Task for multiple-line system using
   * Windows + DDE. Sends commands from upper layers to Server
   * Task (via DDE), and returns events to the upper layers.
   *
   * $Log: /VosysAccess/core/ipc/IPC_WD_C.C $  
   ******************************************************************************/

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

#include <windows.h>
#include <windowsx.h>
#include <dde.h>

#include "OS.h"
#include "dVR.h"
#include "Command.h"
#include "IPC.h"

/*frequency at which to poll for incoming DDE messages (events)*/
#define RESPONSE_POLL_DELAY 200    /* msec between polls */
#define MY_TIMER_ID 1

// The timeout value for client to wait for VosysAccess server to
// respond with any event.
#define CLIENT_WAIT_LIMIT 0         // wait forever

/*
 * Information about state of each active task.
 * This data is shared by all client processes, but each process
 * looks only at the data for its line.
 * Processes find a free slot by looking at the bInUse field.
 */

struct line_t {
    BOOL bInUse;    // is this table slot in use?
unsigned int hTask; // Windows task handle

#define EVENTQ_SIZE 10

struct event eventq[EVENTQ_SIZE];
int nBQNExtFree; // next empty slot
int nBQNExtUnread; // next full (unread) slot

HWND hwndClientDDE; // hidden client window (in this process)
HWND hwndServerDDE; // hidden server window (in server process)

BOOL bConvInTerminateState;
BOOL bCWinCreatedInIPC; // client window created in IPC layer?

UINT timerID;
char szCwClassName[40]; // name of client window class

static struct line_t *line;

// has this layer been initialized yet?
static boolean bInitialized = FALSE;

/***************************
// Find line table slot that is being used by task handle hTask

static int FindSlot (const unsigned int hTask, // task handle (Windows only)
struct line_t **psp) // ptr to ptr to slot in line

done: return nRetVal;

/*****************************/
static
int wait_for_wmessage(
struct line_t * const sp, // ptr to slot in line table
const UINT wmsgtype, /* WM* value */
const int timelimit, /* msec */
MSG *theMsg /* message gotten */
)
{
    unsigned long endtime;
    BOOL notquit;
    int nVRetVal; // Voysys return value
    BOOL bSuccess;
    int nSHuffleCount;

    sp->timerID = SetTimer ( sp->hwndClientDDE, MY_TIMER_ID,
        RESPONSE_POLL_DELAY, NULL );
    if (sp->timerID == 0) {
        nVRetVal = RC_OSErr0r;
        goto done;
    }

    endtime = GetCurrentTime() + timelimit;
    nSHuffleCount = 0;
    while ( GetCurrentTime() < endtime ) {
        notquit = GetMessage ( theMsg,
            (HWND)NULL, // any window
            0, // all messages
            0 );

        if (theMsg->hwnd != sp->hwndClientDDE) {
            // not for us; send to owner
            TranslateMessage ( theMsg ); // Translates virtual

        DispatchMessage ( theMsg ); // Dispatches message

        continue;
        }
    if (theMsg->message == WM_TIMER) {
        // ignore; we just want this to send us around the loop again
        continue;
        }

    if (theMsg->message != wmsgtype) {
        // not the one we are waiting for; stuff back into queue
        // reorders events
        bSuccess = PostMessage(
            sp->hwndClientDDE,
            theMsg->message,
            theMsg->wParam,
            theMsg->lParam
        );

        if (!bSuccess)
            Debug ( 1, sprintf ( Debug_buf,
                "IPC_WD_C/wait_for_wmessage: lost message 0x%04X", theMsg->message ) );
    if (theMsg->message == WM_DDE_TERMINATE) {
        // remove timer */
        KillTimer ( sp->hwndClientDDE, MY_TIMER_ID );
        // server has gone away
        sp->hwndServerDDE = (HWND)NULL;
        nVRetVal = RC_SERVERSHUTDOWN;
goto done;
}
nShuffleCount++;
if (nShuffleCount > 20) {
    /* remove timer */
    KillTimer (sp->hwndClientDDE, MY_TIMER_ID);
    nVRetVal = RC_INTERNAL;
    goto done;
}
continue;

// got it
/* remove timer */
KillTimer (sp->hwndClientDDE, MY_TIMER_ID);
nVRetVal = RC_SUCCESS;
_goto done;

/* remove timer */
KillTimer (sp->hwndClientDDE, MY_TIMER_ID);
nVRetVal = RC_TOTAL_TIMEOUT;

done:
    return nVRetVal;
}

/******************************************************************************

FUNCTION: ClientReceiveData

PURPOSE: Called when client application receives WM_DDE_DATA message.

*******************************************************************************/

static int ClientReceiveData ( 
    struct line_t * const sp, /* ptr to slot in line table 
    const LONG 1Param 
    

    DDEDATA FAR  *lpDDEData;
    BOOL bRelease;
    BOOL bAck;
    int nVRetVal; // Voysys return value
    boolean bSuccess;
    HGLOBAL GFXRetVal; // return value from GlobalFree

    bAck = TRUE;

    if (sp->bConvInTerminateState) { /* Terminate in progress: 
            do not receive data */
        GFXRetVal = GlobalFree ( LOWORD (1Param)); // hData
        if (GFXRetVal != (HGLOBAL)NULL)
            nVRetVal = RC_INTERNAL;
            goto done;
    }

    if (!(lpDDEData = (DDEDATA FAR *) GlobalLock (LOWORD(1Param))))
    
    bSuccess = PostMessage (}
if (!bSuccess)
    Debug ( i, sprintf ( Debug_buf,
        "IPC_WD_C/ClientReceiveData:
        PostMessage(hWndServerDDE 0x%04X)
        failed", sp->hwndServerDDE ) );
#endif

/* check for correct length */
if ( mlnq != sizeof ( struct event ) ) {
    sprintf ( info,
        "got msg with bad length %d (should be %d)",
        mlnq, sizeof ( struct command )
    );

    if ( logging_on )
        print_info ( -1, info, /* not associated with particular line */
            logfile /* info to print */
        );

    if ( tracing_on )
        print_info ( -1, info, /* not associated with particular line */
            tracefile /* info to print */
        );

    bSuccess = PostMessage ( sp->hwndServerDDE,
        WM_DDE_ACK,
        sp->hwndClientDDE,
        MAKELONG ( 0, 0 )
    );
    /* Negative ACK */

    if (!bSuccess)
        nVRetVal = RC_INTERNAL; /* PostMessage() failed */
}

goto done;
#endif

/* do something with data received */
/* put event in queue to be sent up */
sp->eventq[sp->nEQNextFree] = *( ( struct event * ) ( lpDDEData->Value ) );
sp->nEQNextFree++;
if ( sp->nEQNextFree >= EVENTQ_SIZE )
    sp->nEQNextFree = 0;

if ( lpDDEData->fAckReq ) {
    /* return ACK or NACK */
    bSuccess = PostMessage ( sp->hwndServerDDE,
        WM_DDE_ACK,
        sp->hwndClientDDE,
- 172 -

MAKELONG (( bAck ? 0x8000 : 0 ), 0 );

    if (!bSuccess)
        Debug ( 1, sprintf ( Debug_buf,
"IPC_WD_C/ClientReceiveData: PostMessage(hwndServerDDE 0x%04X)
failed", sp->hwndServerDDE ) );

bRelease = lpDDEData->fRelease;
lpDDEData = NULL;
GlobalUnlock ( LOWORD ( lpParam ) );
if ( bRelease ) {
    GFRetVal = GlobalFree ( LOWORD ( lpParam ) );  // hData
    from Server
    if (GFRetVal != (HGLOBAL)NULL)
        Debug ( 1, sprintf ( Debug_buf,
"IPC_WD_C/ClientReceiveData:
GlobalFree(0x%04X) failed",
LOWORD ( lpParam ) ) );
}

nRetVal = RC_SUCCEED;

done:  return nRetVal;
}

/*******************************************************************************
**FUNCTION: ClientTerminate**
**PURPOSE:  Called when client application receives WM_DDE_TERMINATE message.**
*******************************************************************************/

static int ClientTerminate ( struct line_t * const sp )  // ptr to slot in line table
{
    int nRetVal;  // Voysys return value
    boolean bSuccess;

    if ( (!sp->bConvInTerminateState)
        && (sp->hwndServerDDE != (HWND)NULL) ) {
        /* Server has requested terminate: respond with terminate */
        bSuccess = PostMessage ( sp->hwndServerDDE,
WM_DDE_TERMINATE,
sp->hwndClientDDE,
0L );

        if (!bSuccess)
            sprintf ( Debug_buf,
"IPC_WD_C/ClientTerminate:
PostMessage(hwndServerDDE 0x%04X)
failed", sp->hwndServerDDE ) );

        sp->hwndServerDDE = (HWND)NULL;
    }
nRetVal = RC_SUCCEED;

// done:
return nRetVal;
}

 /**************************************************************************
  * get_message
  **************************************************************************/

static
int get_message ( const struct line_t * sp, // ptr to slot in line table
  struct event *event /* event received */)
{
  MSG msg;
  int nRetVal; // Voysys return value
  int limit; // prevent infinite waiting

  /* wait until there is an event in the queue */
  limit = CLIENT_WAIT_LIMIT;
  while ( sp->nEQNextUnread == sp->nEQNextFree ) {
    /* handle any DDE_DATA messages coming up from server */
    nRetVal = wait_for_wmessage ( sp, WM_DDE_DATA, 10000,
      &msg );
    if ( nRetVal == RC_SUCCEED ) {
      ClientReceiveData ( sp, msg lParam );
      limit = CLIENT_WAIT_LIMIT;
    } else if (nRetVal != RC_TOTALTIMEOUT) {
      goto done;
    } else {
      // timeout only if limit != 0 ( i.e., not to wait forever )
      if ( limit ) {
        limit--;
        if (limit <= 0) {
          nRetVal = RC_SERVERCOMMFAIL;
          goto done;
        }
      }
    }

    /* get event from queue */
    *event = sp->eventq[sp->nEQNextUnread];
    sp->nEQNextUnread++;
    if ( sp->nEQNextUnread >= EVENTQ_SIZE )
      sp->nEQNextUnread = 0;
  }

  nRetVal = RC_SUCCEED;

  done:
  return ( nRetVal );
- 174 -

PURPOSE: Send poke message to server.

Tricky: We don't care about the "item name", so we will violate DDE protocol by sending 0 for the name.

static int SendPoke (  
   struct line_t * const sp, // ptr to slot in line table  
   const void  * const ItemValue,  
   const int   ItemSize  
)  
{
   HANDLE hPokeData;  
   DDEPOKE FAR *lpPokeData;  
   int nVRetVal; // Voysys return value  
   MSG msg;  
   boolean bSuccess;  
   HGLOBAL GFRetVal; // return value from GlobalFree

   if ( sp->hwndServerDDE == (HWND)NULL ) {  
      nVRetVal = RC_SERVERSHUTDOWN;  
      goto done;
   }

   /* Allocate size of DDE data header, plus the data */
   hPokeData = GlobalAlloc ( (GMEM_MOVEABLE | GMEM_DDESHARE ),
                           (LONG) sizeof ( DDEPOKE ) + ItemSize );
   if ( hPokeData == (HANDLE)NULL ) {  
      nVRetVal = RC_RAMFULL;  
      goto done;
   }

   lpPokeData = ( DDEPOKE FAR *) GlobalLock ( hPokeData );
   if ( lpPokeData == (DDEPOKE FAR *)NULL ) {
      GFRetVal = GlobalFree ( hPokeData );
      if (GFRetVal != (HGLOBAL)NULL)  
         nVRetVal = RC_OSError;
      goto done;
   }

   lpPokeData->fRelease = TRUE; /* destination should free mem */
   lpPokeData->cfFormat = CF_TEXT;

   memcpy ( lpPokeData->Value, ItemValue, ItemSize );
   lpPokeData = NULL;
   GlobalUnlock ( hPokeData );

   bSuccess = PostMessage (  
      sp->hwndServerDDE,  
      WM_DDE_POKE,  
      sp->hwndClientDDE,  
      MAKELONG ( hPokeData, 0 )  
   );

   if (!bSuccess) {
      Debug ( 1, sprintf ( Debug_buf, "IPC_WD_C/SendPoke:  
                       PostMessage(hwndServerDDE 0x%04X) failed",  
                      sp->hwndServerDDE ) );
      GFRetVal = GlobalFree ( hPokeData );
      if (GFRetVal != (HGLOBAL)NULL)  
         Debug ( 1, sprintf ( Debug_buf, "IPC_WD_C/SendPoke:  
                              GlobalFree(0x%08X) failed", hPokeData ) );
   }

   return nVRetVal;

}
nRetVal = RC_SERVERCOMMPFAIL;
} else {
    nRetVal = wait_for_wmessage ( sp, WM_DDE_ACK, 10000,
                                &msg );
    if ( nRetVal != RC_SUCCEEDED )
        nRetVal = RC_SERVERCOMMPFAIL;
}

done:
    return nRetVal;
}

/****************************
* send_message
****************************/

static int send_message (struct line_t * const sp, // ptr to slot in line table
                          const struct command cmd /* command to send */)
{
    int retval;
    retval = SendPoke (sp,
                        ( void * ) &cmd,
                        sizeof ( cmd )
                        );
    //done:
    return ( retval );
}

FUNCTION: SendTerminate
PURPOSE: Send terminate message to server.
This is called when client decides to terminate
        conversation.

static int SendTerminate (struct line_t * const sp /* ptr to slot in line table */)
{
    boolean bSuccess;
    sp->bConvInTerminateState = TRUE;
    if (sp->hwndServerDDE != (HWND)NULL) {
        bSuccess = PostMessage (sp->hwndServerDDE,
                                WM_DDE_TERMINATE,
                                sp->hwndClientDDE,
                                0L
                                );
    } else {
        bSuccess = PostMessage (sp->hwndClientDDE,
                                WM_DDE_TERMINATE,
                                sp->hwndServerDDE,
                                0L
                                );
    }
    return ( bSuccess );
}

FUNCTION: CloseDialog
PURPOSE: Close DDE dialog.
This is called when client decides to close DDE
        dialog.
if (!bSuccess) {
    sprintf (Debug_buf, *IPC_WD_C/SendTerminate:
              PostMessage(hWndServerDDE, 0x04X)
    failed", hWndServerDDE ));
    hWndServerDDE = (HWND)NULL;
    // there will not be an acknowledge from server
}
}
return RC_SUCCEEDED;

Function: DDEWndProc

Purpose: Handles all DDE messages received by the client application. Tricky: should be virtually identical to
ipcProcessEvent. This function is used if client DDE window was
created by this layer; ipcProcessEvent is used if window created
by higher level.

******************************************************************************

long FAR PASCAL __export DDEWndProc ( HWND hwnd,
                                                UINT message,
                                                WPARAM wParam,
                                                LPARAM lParam )
{
    int nSlot;        // slot in line table
    struct line_t *sp; // ptr to slot in line table

    // find slot in the line table by using hwnd
    for (nSlot=0 ; nSlot<MAX_AL ; nSlot++) {
        if (line[nSlot].bInUse 
            && (line[nSlot].hwndClientDDE == hwnd))
            break;
    }
    if (nSlot == MAX_AL) {
        Debug ( 5, sprintf (Debug_buf, *IPC_WD_C/DDEWndProc: 
                             couldn't find slot in line table* ) );
        return (DefWindowProc (hwnd, message, wParam, lParam )
    }
    sp = &line[nSlot];

    switch (message) {
        case WM_DDE_ACK: /* shouldn't happen */
            return (0L);
        case WM_TIMER:
            return (0L);
        case WM_DDE_DATA:
            ClientReceiveData (sp, lParam);
            return (0L);
    }
}

case WM_DDE_TERMINATE:
    ClientTerminate ( sp );
    return ( 0L );

default:
    return ( DefWindowProc ( hwnd, message, wParam, lParam ) );
}

/******************************************************************************

FUNCTION: ipc_ProcessEvent

PURPOSE: Handles all async DDE messages received by the client application. Tricky: should be virtually identical to DDEWndProc. This function is used if client DDE window was created by higher level; DDEWndProc is used if window was created by this layer.

*******************************************************************************/

int ipc_ProcessEvent ( const unsigned int hTask, // task handle (Windows only)
                        const struct event event // event from above )
{
    int nRetVal; // Voysys return value
    UINT message; // component of Windows message
    WPARAM wParam; // component of Windows message
    LPARAM lParam; // component of Windows message
    struct line_t *sp; // ptr to slot in line table

    nRetVal = RC_SUCCEEDED;

    if ( ( event.nEvent != EV_DDE ) ) {
        nRetVal = RC_INTERNAL;
        goto done;
    }

    // find slot in the line table
    nRetVal = FindSlot ( hTask, &sp );
    if ( ( nRetVal != RC_SUCCEEDED )
        goto done;

    // extract fields from VA form to Windows form
    message = ( UINT ) ( event.dde[0] );
    wParam = ( WPARAM ) ( event.dde[1] );
    lParam = ( LPARAM ) ( event.dde[2] );

    switch ( message ) {
    case WM_DDE_ACK:
        goto done;

    case WM_DDE_DATA:
        ClientReceiveData ( sp, lParam );
        goto done;

    case WM_DDE_TERMINATE:
        ClientTerminate ( sp );
        goto done;
    }
default:
    nVRetVal = RC_INTERNAL;
goto done;
}
done:
    return ( nVRetVal );
}

/**************************************************************************/

FUNCTION: SendInitiate

PURPOSE: Sends initiate message to all windows. By the time this function
         returns, all servers matching the app/topic will have
         temporarily registered the new conversations. If more than one
         server responded, then this client application asks the user
         which conversation to keep; all other conversations will then be
         terminated. This function returns the handle of the hidden DDE
         window used to initiate the conversation with server(s).

Note: Deletes the atoms it creates and the ones sent to it.

/**************************************************************************/

static HWND SendInitiate ( struct line_t * const sp,  // ptr to slot in line table
                          const char *szApplication,
                          const char *szTopic)
{
    ATOM atomApplication;
    ATOM atomTopic;
    int nVRetVal;     // Voysys return value
    MSG mesg;
    ATOM GDARetVal;  // GlobalDeleteAtom return value
    nVRetVal = RC_SUCCEED;

    atomApplication = GlobalAddAtom ( (LPSTR) szApplication );
    if (atomApplication == 0) {
        nVRetVal = RC_OSERROR;
        goto done;
    }

    atomTopic = GlobalAddAtom ( (LPSTR) szTopic );
    if (atomTopic == 0) {
        Debug ( 1, sprintf ( Debug_buf, "IPC_WD/C/SendInitiate: GlobalAddAtom failed" ));
        GDARetVal = GlobalDeleteAtom ( atomApplication );
        if (GDARetVal != 0)
            Debug ( 1, sprintf ( Debug_buf, "IPC_WD/C/SendInitiate: GlobalDeleteAtom(0x%x) failed", atomApplication ));
        nVRetVal = RC_OSERROR;
    }
    goto done;
}
SendMessage(
    ( HWND ) - 1,
    WM_DDE_INITIATE,
    sp->hwndClientDDE,
    MAKELONG ( atomApplication, atomTopic )
);

GDARetVal = GlobalDeleteAtom ( atomApplication );
if (GDARetVal != 0)
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_C/SendInitiate:
GlobalDeleteAtom(0x%04X) failed",
    atomApplication ) );
GDARetVal = GlobalDeleteAtom ( atomTopic );
if (GDARetVal != 0)
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_C/SendInitiate:
GlobalDeleteAtom(0x%04X) failed",
    atomTopic ) );

nVRetVal = wait_for_wmessage ( sp, WM_DDE_ACK, 10000, &msg );
if (nVRetVal != RC_SUCCEEDED ) {
    nVRetVal = RC_NOSERVER;
    goto done;
}

sp->hwndServerDDE = msg.wParam;

// delete atomApplicationReturn and atomTopicReturn from Server
GDARetVal = GlobalDeleteAtom ( LOWORD ( msg.lParam ) );
if (GDARetVal != 0)
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_C/SendInitiate:
GlobalDeleteAtom(0x%04X) failed",
    LOWORD ( msg.lParam ) ) );
GDARetVal = GlobalDeleteAtom ( HIWORD ( msg.lParam ) );
if (GDARetVal != 0)
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_C/SendInitiate:
GlobalDeleteAtom(0x%04X) failed",
    HIWORD ( msg.lParam ) ) );

done:
    return ( nVRetVal );

/***********************/

int  ipc_Initialize(
    const unsigned int  hTask, // task handle (Windows only)
    const unsigned int  hInstance, // instance handle
    const unsigned long hWnd // hidden window handle (Windows only)
)
{
    int     nVRetVal;   // Voysys return value
    WNDCLASS wc;
    ATOM    wrval;
    int     nSlot;     // slot in line table
    struct line_t *sp; // ptr to slot in line table

    nVRetVal = RC_SUCCEEDED;

if (!bInitialized) {
    line = (void *) GlobalAllocPtr(
            GMEM_MOVEABLE | GMEM_ZEROINIT,
            (MAX_AL + sizeof(struct line_t))
        );

    if ( line == NULL ) {
        nRetVal = RC_RAMPULL;
        goto done;
    }
}

bInitialized = TRUE;

// find slot in the line table
nRetVal = FindSlot( hTask, &sp );

if ( nRetVal == RC_SUCCEED ) {
    nRetVal = RC_ALREADYINITIALIZED;
    goto done;
}

// find a free slot in the line table
for (nSlot=0 ; nSlot<MAX_AL ; nSlot++) {
    if (!line[nSlot].bInUse)
        break;
}

if (nSlot >= MAX_AL) {
    Debug ( 5, sprintf ( Debug_buf, "IPC_WD.C/ipc_Initlize:
            line table is full" ));
    nRetVal = RC_SWLIMIT;
    goto done;
}

sp = &line[nSlot];
sp->bInUse = TRUE;
sp->hTask = hTask;

if ( hWnd != ( long ) NULL ) {

    // client window was created up in Database Interface layer
    sp->hwndClientDDE = ( HWND ) hWnd;
    sp->bCWinCreatedInIPC = FALSE;
}

else {

    // we have to create client window in this layer
    wc.style = 0;
    wc.lpfnWndProc = DDEWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = ( HICON ) NULL;
    wc.hCursor = ( HCURSOR ) NULL;
    wc.hbrBackground = ( HBRUSH ) NULL;
    wc.lpszMenuName = NULL;

    sprintf ( sp->szCWClassName, "ClientDDEWndClass", GetCurrentTask ( ) );

    wc.lpszClassName = sp->szCWClassName;

    wrval = RegisterClass ( &wc );

    if ( wrval == 0 ) {
        sp->bInUse = FALSE;
nRetVal = RC_OSErr0r;
goto done;
}
sp->hwndClientDDE = CreateWindow(  
    sp->szCWClassName,  
    "ClientDDE",  
    0, /* not visible */  
    0, 0, 0, 0, /* no position or dimensions */  
    (HWND) NULL, /* no parent */  
    (HWND) NULL, /* no menu */  
    (HINSTANCE) hInstance,  
    NULL  
);
if ( !sp->hwndClientDDE ) {
    Debug ( 1, printf (  
        "IPC_WD_C/ipc_Initialize:  
        CreateWindow failed hwndClient  
        0x%04X", sp->hwndClientDDE );
    sp->bInUse = FALSE;
    nRetVal = RC_INTERNAL;
    goto done;
}
sp->bCWinCreatedInIPC = TRUE;
}
}
nRetVal = SendInitiate (  
    sp,
    VA_DDE_SERVER_APP_NAME,
    VA_DDE_TOPIC
);
done:
    return ( nRetVal );

UPPORTED
*************/
int ipc_Shutdown (  
    const unsigned int hTask // task handle (Windows only)  
)  
{
    int nRetVal; // Voysys return value
    int wrval;
    int nSlot; // slot in line table
    struct line_t *sp; // ptr to slot in line table
    if ( !hInitialized ) {
        nRetVal = RC_NOTINITIALIZED;
        goto done;
    }
    // find slot in the line table
    nRetVal = FindSlot( hTask, &sp );
    if ( nRetVal != RC_SUCCEEDED )
        goto done;
    nRetVal = SendTerminate(sp);
    if ( sp->bCWinCreatedInIPC ) {

// client window was created in this layer
wrval = DestroyWindow ( sp->hwndClientDDE );
wrvall = UnregisterClass ( sp->szCClassName, sp->hTask );

sp->nEQNextFree = 0;
sp->nEQNextUnread = 0;
sp->hwndClientDDE = (HWND)NULL;
sp->hwndServerDDE = (HWND)NULL;
sp->bConvInTerminateState = FALSE;
sp->timerID = 0;
sp->bInUse = FALSE;

// if there are no more used slots, really shut down
for (nSlot=0; nSlot<MAX_AL; nSlot++) {
  if (line[nSlot].bInUse)
    goto done;
}

GlobalFreePtr ( line );
line = NULL;
initialised = TRUE;

done:
return ( nVRetVal );

/*****************************/

int ipc_Do_Command ( const unsigned int hTask, // task handle (Windows only)
  const struct cmd, // command to execute
  struct event *event // result of command
)
{
  int nVRetVal; // VoySys return value
  struct line_t *sp; // ptr to slot in line table

  if (!NO_DEBUGGING_AT_ALL
    if ( Debug_flag >= 2 )
      print_command ( cmd, debugfile );
  #endif

  if (!bInitialized) {
    nVRetVal = RC_NOTINITIALIZED;
    goto done;
  }

  event->nEvent = EV_NONE;
  event->nLine = cmd.nLine;
  event->C_error = 0;
  event->D_ERROR = 0;
  event->DOS_ERROR = 0;
  event->string[0] = '\0';
  event->filename[0] = '\0';

  // find slot in the line table
  nVRetVal = FindSlot ( hTask, &sp );
  if (nVRetVal != RC_SUCCESS)
    goto done;
  // execute command
  nVRetVal = cmd.function ( event );
  // return result
  return ( nVRetVal );

  // dispose of event
  dispose_event ( event );

  // dispose of line
  dispose_line ( line );

  // dispose of slot
  dispose_slot ( &sp );

  // dispose of cmd
  dispose_cmd ( cmd );

  // dispose of hTask
  dispose_hTask ( hTask );
nRetVal = send_message ( sp, cmd );
if ( nRetVal < RC_SUCCEEDED ) {
    event->result = nRetVal;
    goto done;
}

/* wait for event back */
if ( nRetVal < RC_SUCCEEDED ) {
    event->result = nRetVal;
    goto done;
}

done:

#if !NO_DEBUGGING_AT_ALL
    if ( Debug_flag >= 2 )
        print_event ( *event, debugfile );
#endif

return ( event->result );

/*****************************/
APPENDIX F
IPC WD S.c

Description:
- Server Task for multiple-line system using Windows + DDE.
- Accepts commands from Client Tasks (sent via DDE).
- Calls Driver Interface layer to execute commands on phone lines, and returns events to the Line Tasks (via DDE).

Usage:
- IPC_WD_S [ /debugN ] [ /ldhiX ] [ /ldsiXX ]

$Log: /VoysAccess/core/ipc/IPC_WD_S.c $

#include <ctype.h>
#include <dos.h>
#include <fcntl1.h>
#include <io.h>
#include <sys\stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <windows.h>
#include <dde.h>
#include <mmystem.h>

#define IN_MAIN 1
#include "OS.h"
#include "dVR.h"
#include "Command.h"
#include "IPC.h"
#include "TM.h"
#include "srv_wrc.h"

/**************************************************************************
/* default interrupt level and number for line driver card */
#define DEF_LD_HW_IRQ 3
#define DEF_LD_SW_INT 0x62

/* frequency at which to poll for events from below */
/* only polls on lines that have a command in progress */
#define EV_POLL_DELAY 400  /* msec between polls */

/**************************************************************************
/* interrupt level and number for line driver card */
static int ld_hw_irq = DEF_LD_HW_IRQ;
static int ld_sw_int = DEF_LD_SW_INT;
/* filename extensions */
static char promptext[4];  /* extension for prompt files */
static char ifileext[4];   /* extension for indexed prompt files */

struct line_t {
    struct command cmd;  // command
    boolean bCmdInProgress;  // is there a command in progress?
    boolean bConnected;   // is there a task attached to line?
    HWND serverwnd;      // server window handle for line
    HWND clientwnd;      // client window handle for line
    boolean bTestMode;   // is the test mode on for this line?
}

/* stuff set from dvrc_setdir commands */
/* dirs voice files are in */
/* tricky: these names have '\' on the end */
char dirname[MAX_DIRS][MAX_DN+1];

};

static struct line_t line[MAX_AL];

/* stuff used to generate unique user file names */
static int next_user_filename;

#define USER_FRAME_FORMAT "V%07d"

#define WINMAIN_TIMER_ID 1
#define SERVER_TITLE "voysAccess Server"

static boolean bShuttingDown = FALSE;
/* variables used by all functions in this file */
/* they are set to point to current line[] */
static struct line_t *lp = 0;  /* ptr to "line[]" structure */

/************************************
// DDR conversation stuff
/* Maximum values */

#define APP_MAX_SIZE     18
#define TOPIC_MAX_SIZE   11
#define ITEM_NAME_MAX_SIZE 8
#define ITEM_VALUE_MAX_SIZE 8
#define CONV_MAX_COUNT   MAX_AL

typedef struct CONV {
    HWND hWndServerDDE;
    HWND hWndClientDDE;
    BOOL bInClientRequestedTerminate;
};
static struct CONV Conv[CONV_MAX_COUNT];

static int nConvCount = 0;

#define DEF_ACK_TIME_OUT_MILLISecs 10000
#define PEEK_MSG_COUNT 200

static int nAckTimeOut;
static BOOL bTerminating = FALSE;

/******************************************************************************
 /* Global data */
/******************************************************************************

static HANDLE hWndMain;
static HANDLE hInst;
static RECT rect;
static short nCharW, nCharH;

/******************************************************************************
 * parse_arguments - parse command-line
******************************************************************************

static int parse_arguments ( const char *cmdline )
{
    boolean bad_args;
    const char *p;

    bad_args = FALSE;

    p = cmdline;
    while ( *p != '\0' )
    {
        switch ( *p )
        {
            case '/':
                p++;
                if ( strcmp ( p, "debug", 5 ) == 0 )
                {
                    /* note: only support single digit, 0-9 */
                    if ( isdigit ( *( p + 5 ) ) )
                    {
                        bad_args = TRUE;
                    }
                }
                Debug_flag = ( ( *( p + 5 ) ) - '0' );
                break;
        }
    }
p += 6;
Debug ( l,   sprintf ( Debug_buf,
    "IPC_WD_S/parse_arguments: 
    Debug_flag %d", Debug_flag ) );
}
else if ( strncmp ( p, "ldbh", 4 ) == 0 )
{
    /* note: only support single digit, 0-9 */
    if ( !isdigit ( *( p + 4 ) ) )
    {
        bad_args = TRUE;
    }
    ld_hw_irq = ( ( *( p + 4 ) ) - '0' );
    p += 5;
}
else if ( strncmp ( p, "ldsi", 4 ) == 0 )
{
    /* note: only support two hex digits */
    if ( !isxdigit ( *( p + 4 ) ) )
    {
        bad_args = TRUE;
    }
    if ( !isxdigit ( *( p + 5 ) ) )
    {
        bad_args = TRUE;
    }
    if ( isalpha ( *( p + 4 ) ) )
        ld_sw_int = ( tolower ( *( p + 4 ) ) - 'a' + 10 ) * 16;
    else
        ld_sw_int = ( ( *( p + 4 ) ) - '0' ) * 16;
    if ( isalpha ( *( p + 5 ) ) )
        ld_sw_int += ( tolower ( *( p + 5 ) ) - 'a' + 10 );
    else
        ld_sw_int += ( ( *( p + 5 ) ) - '0' );
    p += 6;
}
else
{
    bad_args = TRUE;
}
break;
case '\':
case '\t':
p++; break;
default:
p++;
    bad_args = TRUE;
break;
}

if ( bad_args )
{
    return 1;
}

return 0;

/* function parse_arguments */
void ShowTestMode ( 
  void
) 
{
  HDC  hDC;
  char szBuffer[30];
  short nY;
  int nLine;
  
  hDC = GetDC ( hWndMain );
  SelectObject ( hDC, GetStockObject ( SYSTEM_FIXED_FONT ) );
  for ( nLine = 0; nLine < MAX_AL; nLine++ )
  {
    if ( line[nLine].bTestMode )
    {
      nY = nLine * nCharH;
      TextOut ( hDC, 0, nY, szBuffer, wcsprintf ( szBuffer, "Line #%c is in Test Mode.", (char) ( (int) '0' - nLine ) ) );
    }
  }
  ReleaseDC ( hWndMain, hDC );
  // ValidateRect ( hWndMain, NULL );
} /* function ShowTestMode */

FUNCTION: SendData
PURPOSE: Send data to client.

Tricky: We don't care about the "item name", so we will violate DDE protocol by sending 0 for the name.

static int SendData ( 
  const HWND hWndServerDDE, 
  const HWND hWndClientDDE, 
  const void *ItemValue, 
  const int ItemSize, 
  const BOOL bAckRequest, 
  const BOOL bRequestData 
) 
{
  HANDLE hData;
  DDEDATA FAR *lpData;
  int nVRetVal; // Voyeys return value
  boolean bSuccess;
  HGLOBAL GFRetVal; // return value from GlobalFree
  
  /* Allocate size of DDE data header, plus the data. */
hData = GlobalAlloc (GMEM_MOVEABLE | GMEM_DDESHARE,
    (LONG) sizeof (DDEDA T ) + ItemSize );
if ( hData == (HANDLE)NULL ) {
    nVRetVal = RC_RAMFULL;
    goto done;
}
lpData = (DDEDA T FAR *) GlobalLock (hData);
if ( lpData == NULL ) {
    Debug ( 1, sprintf ( Debug_buf, "IPC_W D/S/SendData:
    GlobalLock failed" ));
    G FRetVal = GlobalFree (hData );
    if ( G FRetVal != (HGLOBAL)NULL)
        Debug ( 1, sprintf ( Debug_buf, "IPC_W D/S/SendData:
            GlobalFree(0x%04X) failed", hData ) );
    nVRetVal = RC_OSERROR;
    goto done;
}
lpData->fAckReq = bAckRequest;
lpData->cF rmat = CP_TEXT;
lpData->fResponse = bRequestData;
lpData->fRelease = TRUE; /* client releases memory */
memcpy ( lpData->Value, ItemValue, ItemSize );
lpData = NULL;
GlobalUnlock (hData);
bSuccess = PostMessage ( 
    hWndClientDDE,
    WM_DDE_DATA,
    hWndServerDDE,
    MAKELONG (hData, 0)
);
if (!bSuccess) {
    /* postmessage failed, so we have to deallocate data */
    G FRetVal = GlobalFree (hData );
    if ( G FRetVal != (HGLOBAL)NULL)
        Debug ( 1, sprintf ( Debug_buf, "IPC_W D/S/SendData:
            GlobalFree(0x%04X) Failed", hData ) );
    nVRetVal = RC_OSERROR; /* !!! */
    goto done;
} /* client will deallocate data */
nVRetVal = RC_SUCCEED;
done:
return nVRetVal;
} /* function sendData */

FUNCTION: AtLeastOneConvActive

PURPOSE: Used during termination of application, to
determine whether any conversations are still active
while the conversations are being terminated.

******************************************************************

static
BOOL AtLeastOneConvActive (  
  void
)  
{  
  return ( nConvCount ? TRUE : FALSE );  
}  
/* function AtLeastOneConvActive */

FUNCTION: FindConv

PURPOSE: Find the conversation for a specified server DDE window.

******************************************************************************

static
struct CONV *NEAR FindConv (  
  const HWND  hWndServerDDE
)  
{  
  struct CONV  *pConv;  
  int     nConvIndex;
  for ( nConvIndex = 0, pConv = Conv;  
       nConvIndex < nConvCount;  
       nConvIndex++, pConv++ )  
  {  
    if ( pConv->hWndServerDDE == hWndServerDDE )  
      return ( pConv );  
  }  
  return ( NULL );  
}  
/* function FindConv */

FUNCTION: GetNextConv

PURPOSE: Get next client in list of conversations. To get the first hWndServerDDE in the conversation list, pass in a NULL value for hWndServerDDE.

******************************************************************************

static
HWND GetNextConv (  
  const HWND  hWndServerDDE
)  
{  
  struct CONV  *pConv;  
  int     nConvIndex;
  if ( hWndServerDDE )  
  {  
    for ( nConvIndex = 0, pConv = Conv;  
         nConvIndex < nConvCount;  
         nConvIndex++, pConv++ )  
      return ( pConv );  
  }
{ if ( pConv->hWndServerDDE == hWndServerDDE )
  { if ( ++nConvIndex < nConvCount )
      return ( ++pConv )->hWndServerDDE;
    else
      return ( ( HWND ) NULL );
  }
  return ( ( HWND ) NULL );
}
if ( nConvCount > 0 )
  return ( Conv[0].hWndServerDDE );
else
  return ( ( HWND ) NULL );
} /* function GetNextConv */

******************************************************************************
FUNCTION: AddConv

PURPOSE: Register a new conversation with a client window
******************************************************************************

static
BOOL AddConv ( const HWND hWndServerDDE,
                const HWND hWndClientDDE )
{
  struct CONV *pConv;

  if ( nConvCount >= CONV_MAX_COUNT )
    return ( FALSE );

  if ( FindConv ( hWndServerDDE ) != NULL )
    return ( FALSE ); /* conversation already added */

  pConv = Conv + nConvCount++;
  pConv->hWndServerDDE = hWndServerDDE;
  pConv->hWndClientDDE = hWndClientDDE;
  pConv->hInClientRequestedTerminate = FALSE;
  return ( TRUE );
} /* function AddConv */

******************************************************************************
FUNCTION: GetHwndClientDDE

PURPOSE: Get the hWnd of the client in conversation with a specified server DDE window.
static HWND GetHwndClientDDE ( 
  const HWND hWndServerDDE
) 
{
    struct CONV  *pConv;
    if (! ( pConv = FindConv ( hWndServerDDE ) ) ) 
      return ( ( HWND ) NULL );
    return ( pConv->hWndClientDDE );
} /* function GetHwndClientDDE */

FUNCTION: IsConvInTerminateState
PURPOSE: Terminate whether conversation with specified
client is in process of being terminated.

static BOOL IsConvInTerminateState ( 
  const HWND hWndServerDDE
) 
{
    struct CONV  *pConv;
    if ( pConv = FindConv ( hWndServerDDE ) ) 
      return ( pConv->bInClientRequestedTerminate );
    else
      return ( FALSE );
} /* function IsConvInTerminateState */

FUNCTION: RemoveConv
PURPOSE: Remove conversation from conversation list.

static void RemoveConv ( 
  const HWND hWndServerDDE
) 
{
    struct CONV  *pConv;
    int  nConvIndex;
    for ( nConvIndex = 0, pConv = Conv; 
          nConvIndex < nConvCount; 
          nConvIndex++, pConv++ )
    {
if ( pConv->hWndServerDDE == hWndServerDDE )
    break;
}
nConvCount--;
while ( nConvIndex < nConvCount )
{
    *pConv = *( pConv + 1 );
    nConvIndex++;
    pConv++;
}
return;
} /* function RemoveConv */

/* *******************************************************************

FUNCTION: SetConvInTerminateState
PURPOSE: Set conversations's terminate state to TRUE.
*******************************************************************/

static
void SetConvInTerminateState ( 
    const HWND hWndDDE 
)
{
    struct CONV  *pConv;
    if ( pConv = FindConv ( hWndServerDDE ) )
        pConv->bInClientRequestedTerminate = TRUE;
    return;
} /* function SetConvInTerminateState */

/* *******************************************************************

FUNCTION: InitAckTimeOut
PURPOSE: Get DDE timeout value from win.ini. Value is in milliseconds.
*******************************************************************/

static
void InitAckTimeOut ( 
    void
)
{
    /* Finds value in win.ini section corresponding to application name */
    nAckTimeOut = GetPrivateProfileInt ( 
        "voysAccess Server", 
        "DDETimeOut", 
        DEF_ACK_TIME_OUT_MILLISEC,
#if defined(TYIN4000)
  "voysrvt.ini" // may need to use different name
  for different server
#else
  "voysrved.ini" // may need to use different name
  for different server
#endif

  return;
}

} /* function InitAckTimeOut */

/**************************************************************************
  * send_event
  *
  * Note: deliberately don't check line[].bConnected, because if we
  * just had a DVRC_DISCONNECT, it will be set to FALSE even though
  * all of the other line[] info is still valid.
  **************************************************************************/

static
void send_event ( const struct event event, /* event to send */
  int *nVRetVal /* return value; 0 == okay */
)
{
  if ( Debug_flag >= 2 )
    print_event ( event, debugfile );

  /* send the resulting message to the other task */
  if (line[event.nLine].serverwnd == (HWND)NULL) {
    *nVRetVal = RC_SERVERCOMMFAIL; // could use better error
code here
    goto done;
  }

  *nVRetVal = SendData ( line[event.nLine].serverwnd,
    line[event.nLine].clientwnd,
    &event,
    sizeof ( event ),
    TRUE, /* request an ACK */
    FALSE /* client did not request this data */
  );

done:
} /* function send_event */

/**************************************************************************
  * do_polling - call line driver to have it check for events.
  **************************************************************************/

static
void do_polling ( void
{
  struct event ev;       // outgoing event allocation
int nRetVal;  // Voysys return value
int nLine;

for ( nLine = 0; nLine < MAX_AL; nLine++ )
{
  if ( line[nLine].bCmdInProgress )
  {
    nRetVal = tm_Do_Command ( line[nLine].cmd,
      FALSE, /* not start of command */
      &ev );
    if ( nRetVal != RC_CMDNOTDONE )
    {
      /* send event back to Client */
      send_event ( ev, &nRetVal
      );
      line[nLine].bCmdInProgress = FALSE;
    }
  }
}

} /* function do_polling */

/******************************************************************************
* do_command - execute a command.
*******************************************************************************/

static void do_command ( 
  struct command * cmd,    // incoming command (same name as
    in IPC_1) 
  const HWND hWndClientDDE, 
  const HWND hWndServerDDE 
)
{
  struct event *event;  // outgoing event (same name as in
    in IPC_1)
  struct event ev;     // outgoing event allocation
  int i;               
  int nRetVal;         // Voysys return value
  int nFRetVal;        // file operation return value
  int nLine;
  char errmsg[140];   
  struct stat statbuf; // buffer of information about a
directory
  char tmpbname[MAX_FN + 5]; // base filename (include the
dot and 3 chars for file
extension
  char tmpfname[MAX_DN + MAX_FN + 5]; // full filename
  (include the dot and 3
chars for file
extension
  boolean foundfile;    // was file found ?

  if ( Debug_flag >= 2 )
    print_command ( cmd, debugfile );

  event = &ev;  /* to match IPC_1 */
nLine = tcmd.nLine;
event->nEvent = EV_NONE;
event->nLine = nLine;
5 event->c_error = 0;
event->DV_error = 0;
event->DOS_error = 0;
event->string[0] = '\0';
10 event->filename[0] = '\0';
event->bInTestMode[nLine] = line[nLine].bTestMode;

if ( ( tcmd.nCommand != DVRC_CONNECT ) &&
    ( tcmd.nCommand != DVRC_SETDIR ) &&
    ( tcmd.nCommand != DVRC_GETSTATE ) &&
    ( !line[nLine].bConnected ) ) {  
15 event->nEvent = EV_NONE; /* lazy */
event->result = RC_NOTCONNECTED;
    nVRetVal = event->result;
goto finish;
}

20 switch ( tcmd.nCommand ) {

25 case DVRC_CONNECT:
    if ( (line[nLine].serverwnd != (HWND)NULL) ) {
1// previous owner did a disconnect, but DDE conversation
1// has not been torn down yet.
        event->result = RC_BUSYLINE;
        nVRetVal = event->result;
        goto finish;
    }
    line[nLine].bConnected = TRUE;
    line[nLine].clientwnd = hWndClientDDE;
    line[nLine].serverwnd = hWndServerDDE;
    line[nLine].bTestMode = FALSE;
    goto senddown;
35 case DVRC_DISCONNECT:
40    line[nLine].bConnected = FALSE;
    line[nLine].bTestMode = FALSE;
    UpdateWindow ( hWndMain );
    /* note: other line[] info still valid, for send_event */
    /* note: DDE conversation still active (will be torn
    // down when get DDE_TERMINATE event). */
45 goto senddown;

50 case DVRC_GETSTATE:
    event->nEvent = EV_GOTSTATE;
    event->has_task[nLine] = line[nLine].bConnected;
    event->bInTestMode[nLine] = line[nLine].bTestMode;
    if ( !line[nLine].bCmdInProgress )
        event->nCommand[nLine] = DVRC_NONE;
    else
        event->nCommand[nLine] = line[nLine].cmd.nCommand;
    if ( !line[nLine].bCmdInProgress )
        goto senddown;
    else
        event->user_off_hook = TRUE; /* can’t tell */
55 event->result = RC_SUCCEEDED;
    nVRetVal = event->result;
    goto finish;

60}
case DVRC_GETSYSSTAT:
    event->nEvent = EV_GETSYSSTAT;
    for ( i = 0; i < MAX_LINES; i++ ) {
        if ( i < MAX_AL ) {
            event->has_task[i] = line[i].bConnected;
            event->bInTestMode[i] = line[i].bTestMode;
        } else {
            event->has_task[i] = FALSE;
            event->bInTestMode[i] = FALSE;
        }
    }
    for ( i = 0; i < MAX_LINES; i++ ) {
        if ( i < MAX_AL ) {
            if ( !line[i].bCmdInProgress )
                event->nCommand[i] = DVRC_NONE;
            else
                event->nCommand[i] = line[i].cmd.nCommand;
        } else {
            event->nCommand[i] = DVRC_NONE;
        }
    }
    event->result = RC_SUCCEEDED;
    goto finish;

    /* TRICKY: FROM HERE ON DOWN, CODE IS IDENTICAL IN 
    * IPC_1 AND IPC_*_S 
    */

    /* command that can be executed in this layer */
    case DVRC_SETDIR:
        event->nEvent = EV_SETDIR;
        if ( ( tcmd.dir_num < 0 ) || ( tcmd.dir_num >= MAX_DIRS ) ) {
            event->result = RC_BADCOMMAND;
            nVRetVal = event->result;
            goto finish;
        }
        if ( strlen ( tcmd.name ) > MAX_DM ) {
            event->result = RC_BADCOMMAND;
            nVRetVal = event->result;
            goto finish;
        }
        /* tricky: even if directory does not exist or is not 
         * accessible, record its name (and return error code, 
         * of course). 
         */
        strcpy ( line[nLine].dirname[tcmd.dir_num], tcmd.name);
        strcat ( line[nLine].dirname[tcmd.dir_num], "\" );
        /* see if dir exists and is readable and writable */
        if defined(E_F)
            nFRetVal = open ( tcmd.name, O_RDONLY );
        if ( nFRetVal >= 0 ) {
            close ( nFRetVal );
            event->result = RC_BADCOMMAND;
            nVRetVal = event->result;
            goto finish;
        }
        else
            event->result = RC_BADCOMMAND;
            nVRetVal = event->result;
            goto finish;

/* eisdir not found in watcom ERRNO.H
   if (errno != EISDIR) {
     event->result = RC_BADCOMMAND;
     nVRetVal = event->result;
     goto finish;
   }
*/

#else

nFRetVal = stat ( tcmd.name, &statbuf );
if ( nFRetVal != 0 ) {
  event->C_error = errno;
  event->DOS_error = _doserrno;
  sprintf ( 
    errmsg,
    "Could not get status of directory '\"",
    tcmd.name
  );
  event->result = RC_BADCOMMAND;
  nVRetVal = event->result;
  goto finish;
}

if ( ( statbuf.st_mode & S_IFDIR ) == 0 ) {
  Debug ( 6, sprintf ( Debug_buf, "IPC: '\"", tcmd.name ) )
  event->result = RC_BADCOMMAND;
  nVRetVal = event->result;
  goto finish;
}

/* apparently directories are never "writable" in DOS */
if ( ( ( statbuf.st_mode & S_IREAD ) == 0 ) ||
    FALSE /* ((statbuf.st_mode & S_IWRITE) ==0 */ ) {
  Debug ( 6, sprintf ( Debug_buf, "IPC: Directory '\"", tcmd.name )
  event->result = RC_BADCOMMAND;
  nVRetVal = event->result;
  goto finish;
}
#endif

event->result = RC_SUCCEEDED;
        nVRetVal = event->result;
        goto finish;

case DVRC_DELETEFILE:
  event->nEvent = EV_DELETEFILE;
  if ( ( tcmd.dir_num < 0 ) || ( tcmd.dir_num >= MAX_DIRS ) )
  {
    event->result = RC_BADCOMMAND;
    nVRetVal = event->result;
    goto finish;
  }

  if ( strlen ( tcmd.play_item.fname ) > MAX_FN )
  {
    event->result = RC_BADCOMMAND;
    nVRetVal = event->result;
    goto finish;
  }

  /* prepend appropriate directory name to filename */
  strcpy ( tmpfname, line[nLine].dirname[tcmd.dir_num]);
- 199 -

strcat ( tmpfname, tcmd.play_item.fname );
if ( !fname_has_extension ( tcmd.play_item.fname ) )
{
    /* append appropriate extension to filename */
    strcat ( tmpfname, "." );
    strcat ( tmpfname, promptext );
}
event->nEvent = EV_DELETEFILE;
nFRetVal = remove ( tmpfname );
if ( nFRetVal < 0 )
{
    Debug ( 6, sprintf ( Debug_buf, "IPC: remove failed \d", nFRetVal ) );
event->result = RC_MOFILE;
event->C_error = errno;
#if !defined(E_F)
event->DOS_error = _doserrno;
#endif
    nVRetVal = event->result;
    goto finish;
}
event->result = RC_SUCCEED;
nVRetVal = event->Result;
goto finish;

/* Note: command that has to be sent down to lower layer */
/*
* tricky: for some commands, change command slightly before
* sending it down.
*/

case DVRTC_RECORD:
    tcmd.dir_num = tcmd.path[0];
    if ( ( tcmd.dir_num < 0 ) || ( tcmd.dir_num >=
        MAX_DIRS ) )
    {
        event->result = RC_BADCOMMAND;
        nVRetVal = event->Result;
        goto finish;
    }
    if ( strlen ( tcmd.play_item.fname ) > MAX_FN )
    {
        event->result = RC_BADCOMMAND;
        nVRetVal = event->Result;
        goto finish;
    }
    /* if upper layer gave filename, use it, else generate */
    if ( tcmd.play_item.fname[0] != '\0' )
    {
        /* prepend appropriate directory name to filename */
        strcpy(tmpfname, line[nLine].dirname[tcmd.dir_num]);
        strcat ( tmpfname, tcmd.play_item.fname );
        if ( !fname_has_extension ( tcmd.play_item.fname ) )
        {
            /* append appropriate extension to filename */
            strcat ( tmpfname, "." );
            strcat ( tmpfname, promptext );
        }
        else
        {
            /* generate unique filename */
            while ( TRUE )
            {
{  
    sprintf(  
        tcmd.play_item.fname,  
        USER_FILENAME_FORMAT,  
        next_user_filename  
    );  
    next_user_filename++;  
    /* prepend appropriate directory name to filename */  
    strcpy( tmpfname,  
        line[nLine].dirname[tcmd.dir_num] );  
    strcat ( tmpfname, tcmd.play_item.fname );  
    strcat ( tmpfname, "\"" );  
    strcat ( tmpfname, promptext );  
    /* see if the file exists */  
    #if defined(E_F)  
    nFRetVal = open ( tmpfname, O_RDONLY );  
    if ( nFRetVal == -1 )  
        break; /* file does not exist */  
    close ( nFRetVal );  
    #else  
    nFRetVal = stat ( tmpfname, &statbuf );  
    if ( nFRetVal != 0 )  
        break; /* file does not exist */  
    #endif  
  }  
  strcpy ( tcmd.play_item.fname, tmpfname );  
Debug ( 6, sprintf ( Debug_buf, "IPC: full filename \"%s\", tcmd.play_item.fname ) );  
  goto senddown;  
  
  case DVRC_ADDPLAY:  
  case DVRC_SETFILE:  
  case DVRC_SETVocab:  
  if ( strlen ( tcmd.play_item.fname ) > MAX_FN )  
    {  
      event->result = RC_BADCOMMAND;  
      nVRetVal = event->result;  
      goto finish;  
    }  
  strcpy ( tmpfname, tcmd.play_item.fname );  
  if ( !fname_has_extension ( tcmd.play_item.fname ) )  
    {  
      /* append appropriate extension to filename */  
      if ( tcmd.nCommand == DVRC_ADDPLAY )  
        {  
          strcat ( tmpfname, "\." );  
          strcat ( tmpfname, promptext );  
        }  
      else if ( tcmd.nCommand == DVRC_SETFILE )  
        {  
          strcat ( tmpfname, "\." );  
          strcat ( tmpfname, ifileext );  
        }  
    }  
    
    Find_File (  
        tmpfname,  
        tcmd.path,  
        line[nLine].dirname,  
        &foundfile,  
        /* was file found? */  
        tmpfname  
    );
if (foundfile)
{
    /* file exists */
    strcpy ( tcmd.play_item.fname, tmpfname );
    if ( tcmd.nCommand == DVRC_ADDPLAY )
        goto senddown;
    /* do same for map file */
    Find_File ( tcmd.name, /* base file name */
                tcmd.path, /* path to look through */
                line[nLine].dirname, /* names of dirs */
                &foundfile, /* was file found ? */
                tmpfname /* full file name */);
    if (foundfile)
    {
        /* file exists */
        strcpy ( tcmd.name, tmpfname );
        goto senddown;
    }
    else
    {
        event->result = RC_NOFILE;
        nRetVal = event->result;
        goto finish;
    }
}
else
{
    event->result = RC_NOFILE;
    nRetVal = event->result;
    goto finish;
}

case DVRC_SETINPUT:
case DVRC_SETOUTPUT:
case DVRC_FAXOUTPUT:
case DVRC_FAXDOC:
case DVRC_FAXSEND:
case DVRC_FAXSETUP:
    goto senddown;

/* Note: command that has to be or not to be sent down to lower layer */
case DVRC_SET:
    switch ( tcmd.nSetGetOpCode )
    {
        case SET_TESTMODE:
            if ( tcmd.dwSetParm == SET_TM_ON )
            {
                line[nLine].bTestMode = TRUE;
                event->bInTestMode[line] = TRUE;
            }
            else
            {
                line[nLine].bTestMode = FALSE;
                event->bInTestMode[line] = FALSE;
            }
            event->result = RC_SUCCEEDED;
            nRetVal = event->result;
    }
    UpdateWindow ( hWndMain );
    goto finish;
- 202 -

} /* end of switch for tcvd.nSetGetOpCode */
goto senddown;

case DVMC_GET:
    switch ( tcvd.nSetGetOpCode ) {
        case GET_FILEFORMATS:
            goto senddown;

        case GET_RECORDINGS:
        case GET_TMPFILES:
            event->nEvent = EV_GET;
            if ( ( tcmd.dir_num < 0 ) || ( tcmd.dir_num >= MAX_DIRS ) ) {
                event->result = RC_BADCOMMAND;
                nVRetVal = event->result;
                goto finish;
            }

            strncpy ( event->cGetParm,
                    line[nLine].dirname[tcmd.dir_num] );
            event->result = RC_SUCCEED;
            nVRetVal = event->Result;
            goto finish;

        } /* end of switch for tcmd.nSetGetOpCode */
goto senddown;

default:
goto senddown;

} /* TRICKY: FROM HERE UP, CODE IS IDENTICAL IN IPC_L AND IPC_L_S */

} /* never get here */

senddown:
    line[nLine].cmd = tcmd;
    line[nLine].bCmdInProgress = TRUE;
    nVRetVal = tm_Do_Command ( tcmd, TRUE, event );

finish:
    if ( nVRetVal != RC_CMDNOTDONE ) {
        /* send event back to Client */
        send_event ( *event, lnVRetVal );

        line[nLine].bCmdInProgress = FALSE;
    }
}
FUNCTION: SendTerminate

PURPOSE: Post terminate message and indicate that conversation is in process of being terminated.

************

static int SendTerminate (
    const HWND hWndServerDDE,
    const HWND hWndClientDDE
)
{
    boolean bSuccess;

    SetConvInTerminateState ( hWndServerDDE );
    bSuccess = PostMessage ( hWndClientDDE, WM_DDE_TERMINATE,
                            hWndServerDDE, 0L );
    if ( bSuccess ) {
        Debug ( 1, sprintf ( Debug.buf, "IPC_WD_S/SendTerminate:
            PostMessage(hWndClientDDE 0x404X) failed", hWndClientDDE ) );
    }
    return RC_SUCCEED;
}

/* function SendTerminate */

FUNCTION: ServerAcknowledge

PURPOSE: Called when server application receives ACK or NACK, or when server receives time out waiting for response to WM_DDE_DATA.

************

static int ServerAcknowledge (
    const HWND hWndServerDDE,
    const HWND hWndClientDDE,
    const LONG lParam
)
{
    char szItemName[ITEM_NAME_MAX_SIZE + 1];
    ATOM GDARetVal;       // GlobalDeleteAtom return value

    if ( ! ( LOWORD ( lParam ) & 0x8000 ) ) {
        Debug ( 1, sprintf ( Debug.buf, "IPC_WD_S/ServerAcknowledge: DDE send data failed" ) );
        /* timeOut after DDE_SEND_DATA */
        if ( HIWORD ( lParam ) != 0 ) {
            GlobalDeleteAtom ( HIWORD ( lParam ), szItemName,
                              ITEM_NAME_MAX_SIZE );
            /* free the data here */
        }
    }
    if ( HIWORD ( lParam ) != 0 ) { /* 0 if time-out, so don’t try to delete */
        GDARetVal = GlobalDeleteAtom ( HIWORD ( lParam ) );
        if ( GDARetVal != 0)
Debug(1, sprintf(Debug_buf,"IPC_WD_S/ServerAcknowledge: GlobalDeleteAtom(0x404) failed", HIWORD ( lParam ) ));

return RC_SUCCEEDED;

} /* function ServerAcknowledge */

FUNCTION: ServerInitiate

PURPOSE: Called when server application receives WM_DDINITIATE message.

Note: Client will delete atoms it created and the ones we create here.

*******************************************************************************/

static int ServerInitiate ( const HWND hWndClientDDE, const LONG lParam ) {
    HWND hWndServerDDE;
    ATOM atomApplicationRcvd;
    ATOM atomTopicRcvd;
    ATOM atomApplicationReturn;
    ATOM atomTopicReturn;
    char szApplication[APP_MAX_SIZE + 1];
    char szTopic[TOPIC_MAX_SIZE + 1];
    int nRetVal;  // Voysys return value
    boolean bSuccess;
    ATOM GDAReturn;

    atomApplicationRcvd = LOWORD ( lParam );
    if ( atomApplicationRcvd != 0 ) {
        GlobalGetAtomName ( atomApplicationRcvd, szApplication, APP_MAX_SIZE );
        if ( strcmpi ( szApplication, VA_DDE_SERVER_APP_NAME ) ) {
            /* application was specified but it wasn’t right */
            nRetVal = RC_BADCOMMAND;
            goto done;
        }
    }

    atomTopicRcvd = HIWORD ( lParam );
    if ( atomTopicRcvd != 0 ) {
        GlobalGetAtomName ( atomTopicRcvd, szTopic, TOPIC_MAX_SIZE );
        if ( strcmpi ( szTopic, VA_DDE_TOPIC ) ) {
            /* topic wasn’t right */
            nRetVal = RC_BADCOMMAND;
            goto done;
        }
    }
}

DONE:
hWndServerDDE = CreateWindow (  
    "voyysServerDDEWndClass",  
    "voyysServerDDE",  
    WS_CHILD,  
    0, 0, 0, 0, /* no position or dimensions */  
    hWndMain, /* parent */  
    ( HMENU ) NULL, /* no menu */  
    hInst,  
    NULL  
);  

if ( hWndServerDDE == (HWND)NULL) {  
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerInitiate:  
        CreateWindow failed" ) );  
    nRetVal = RC_OSERROR;  
    goto done;  
}  

bSuccess = AddConv ( hWndServerDDE, hWndClientDDE );  
if ( !bSuccess ) {  
    DestroyWindow ( hWndServerDDE );  
    nRetVal = RC_SWLIMIT;  
    goto done;  
}  

atomApplicationReturn = GlobalAddAtom (VA_DDE_SERVER_APP_NAME);  
if (atomApplicationReturn == 0) {  
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerInitiate:  
        GlobalAddAtom failed" ) );  
    RemoveConv ( hWndServerDDE );  
    DestroyWindow ( hWndServerDDE );  
    nRetVal = RC_OSERROR;  
    goto done;  
}  

atomTopicReturn = GlobalAddAtom ( VA_DDE_TOPIC );  
if (atomTopicReturn == 0) {  
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerInitiate:  
        GlobalAddAtom failed" ) );  
    GDARetVal = GlobalDeleteAtom ( atomApplicationReturn );  
    if (GDARetVal == 0)  
        Debug (1,sprintf ( Debug_buf,"IPC_WD_S/ServerInitiate:  
            GlobalDeleteAtom(0x%04X) failed",  
            atomApplicationReturn ) );  
    RemoveConv ( hWndServerDDE );  
    DestroyWindow ( hWndServerDDE );  
    nRetVal = RC_OSERROR;  
    goto done;  
}  

// Send ACK to hWndClientDDE from hWndServerDDE  
// supposed to use SendMessage here, but if we do client won't  
// unblock, because it is in SendMessage sending to us!  
bSuccess = PostMessage (  
    hWndClientDDE,  
    WM_DDE_ACK,  
    hWndServerDDE,  
    MAKELONG ( atomApplicationReturn, atomTopicReturn )  
);  
if (!bSuccess) {  
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerInitiate:  
        PostMessage failed" ) );  
    GDARetVal = GlobalDeleteAtom ( atomApplicationReturn );  
    if (GDARetVal == 0)
Debug ( 1, sprintf( Debug_buf, "IPC_WD_S/ServerInitiate:  
GlobalDeleteAtom(0x#04X) failed",
atomApplicationReturn ));

GDARetVal = GlobalDeleteAtom ( atomTopicReturn );
if ( GDARetVal != 0 )
  Debug ( 1, sprintf( Debug_buf, "IPC_WD_S/ServerInitiate:  
GlobalDeleteAtom(0x#04X) failed",
atomTopicReturn ) );

RemoveConv ( hWndServerDDE );
DestroyWindow ( hWndServerDDE );
nVRetVal = RC_OSBERROR;
goto done;
}

nVRetVal = RC_SUCCESS;

done:
  return nVRetVal;
}

/* function ServerInitiate */

/*******************************************************************************/

FUNCTION: ServerPoke

PURPOSE: Called when server application receives WM_DDE_POKE message, which is a command from a client application.

Tricky: Ignore "item name" portion of message; it will be 0.

*******************************************************************************/

static
int ServerPoke ( 
const HWND hWndDDE,          
const HWND hWndClientDDE,     
const LONG lParam
) {
  HANDLE h PokeData;
  DDEpoke FAR *lp PokeData;
  BOOL b Release;
  struct command tc md;     /* incoming command */
  struct event et;           /* event to send */
  char info[80];             /* info message */
  BOOL sendevent;
  int nVRetVal;              // Voysys return value
  boolean bSuccess;
  GLOBAL GRetVal;            // return value from GlobalFree

  hPokeData = LOWORD ( lParam ); /* handle to command */

  if ( !( lpPokeData = ( DDEPOKE FAR * ) GlobalLock ( hPokeData))
    || ( lpPokeData->cfFormat != CF_TEXT ) ) {
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerPoke: data  
lock failed, or wrong data format" ) );
    bSuccess = PostMessage ( 
      hWndClientDDE,  
      WM_DDE_ACK,  
      hWndServerDDE,  
      0 ));
  }

/*******************************************************************************/
- 207 -

MAKELONG ( 0, 0 );    /* negative acknowledgement */
if (!bSuccess) {
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerPoke:
        PostMessage failed" ) );
}

nRetVal = RC_OSERROR;
goto done;

memcpy ( &tcmd, lpPokeData->Value, sizeof ( tcmd ) );
sendevent = FALSE;

/* check for valid line number */
if ( ( tcmd.nLine < 0 ) || ( tcmd.nLine >= MAX_AL ) )
{
    sprintf ( info,
        "got msg with bad line num %d (should be in range
        0, %d-%d)",
        tcmd.nLine,
        0,
        MAX_LINES - 1
    );

    /* send an event back to the sender */
    memset ( &event, 0, sizeof ( event ) );
    event.nEvent = EV_BADCOMMAND;
    event.nLine = -1;
    event.result = RC_INTERNAL;
    sendevent = TRUE;
    goto skipcmd;
}

/* check for task A connected to line but command is from task B */
if ( line[tcmd.nLine].bConnected &&
    ( line[tcmd.nLine].clientwnd != hWndClientDDE ) )
{
    sprintf ( info,
        "got msg for line num %d from task != owner of line",
        tcmd.nLine
    );

    /* send an event back to the sender */
    memset ( &event, 0, sizeof ( event ) );
    event.nEvent = EV_BADCOMMAND;
    event.nLine = tcmd.nLine;
    event.result = RC_BUSYLINE;
    sendevent = TRUE;
    goto skipcmd;
}

// check for server shutting down
if ( bShuttingDown &&
    ( tcmd.nCommand != DVRC_HANGUP ) &&
    ( tcmd.nCommand != DVRC_DISCONNECT ) )
{
    sprintf ( info,
        "got msg for line num %d during shutdown",
        tcmd.nLine
    );
tcmd.nLine
};
/* send an event back to the sender */
memset ( &event, 0, sizeof ( event ) );
event.nEvent = EV_BADCOMMAND;
event.nLine = tcmd.nLine;
event.result = RC_SERVERSHUTDOWN;
sendevent = TRUE;
goto skipcmd;
}
do_command ( tcmd, hWndClientDDE, hWndServerDDE );

skipcmd:
  /* Save value of fRelease, since ptr invalidated by */
bRelease = lpPokeData->fRelease;
lpPokeData = NULL;
GlobalUnlock ( hpokeData );

if ( bRelease ) {
  GFRetVal = GlobalFree ( hpokeData );
  if ( GFRetVal != (HGLOBAL)NULL)
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerPoke: 
                GlobalFree(0x%04X) failed", hpokeData ) );
}

bSuccess = PostMessage ( hWndClientDDE,
  WM_DDE_ACK,
  hWndServerDDE,
  MAKELONG ( 0x8000, 0 ) ); /* positive acknowledgement */

if (!bSuccess) {
  Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/ServerPoke: 
                PostMessage failed" ) );
}

if ( sendevent )
{
  /* send the resulting message to the other task */
  SendData ( hWndServerDDE,
      hWndClientDDE,
      &event,
      sizeof ( event ),
      TRUE, /* request an ACK */
      FALSE /* client did not request this data */
    );
}
nRetVal = RC_SUCCEED;
done:
  return nRetVal;
} /* function ServerPoke */
FUNCTION: ServerTerminate

PURPOSE: Called when server application receives WM_DDE_TERMINATE message.

static int ServerTerminate (
    const HWND hwndServerDDE,
    const HWND hwndClientDDE
) {
    int nLine;
    int nRetVal;  // Voysys return value

    nRetVal = RC_SUCCEED;

    if ( FindConv ( hwndServerDDE ) == NULL )
        goto done;

    RemoveConv ( hwndServerDDE );
    DestroyWindow ( hwndServerDDE );

    for ( nLine = 0; nLine < MAX_AL; nLine++ ) {
        if ( line[nLine].serverwnd == hwndServerDDE ) {
            line[nLine].serverwnd = (HWND)NULL;
            line[nLine].clientwnd = (HWND)NULL;
            goto done;
        }
    }

    nRetVal = RC_INTERNAL;

    done:
    return nRetVal;
} /* function ServerTerminate */

FUNCTION: TerminateConversations

PURPOSE: Processes WM_DESTROY message, terminates all conversations.

static int TerminateConversations ( void ) {
    HWND hwndServerDDE;
    LONG lTimeOut;
    MSG msg;

    /* Terminate each active conversation */
    hwndServerDDE = (HWND) NULL;
    while ( hwndServerDDE = getNextConv ( hwndServerDDE ) ) {
        SendTerminate ( hwndServerDDE, GetHwndClientDDE (}
hWndServerDDE })
}

} /* Wait for all conversations to terminate OR for time out */

lTimeOut = GetTickCount() + (LONG)nAckTimeOut;
while ( PeekMessage(&msg, (HWND)NULL, WM_DDE_FIRST,

WM_DDE_LAST, PM_REMOVE) ) {
  DispatchMessage(&msg);
  if (msg.message == WM_DDE_TERMINATE) {
    if (!AtLeastOneConvActive())
      break;
  }
  if (GetTickCount() > (DWORD)lTimeOut)
    break;
}

return RC_SUCCEEDED;

} /* function TerminateConversations */

FUNCTION: DDEWndProc

PURPOSE: Handles all DDE messages received by the server application.

long FAR PASCAL __export DDEWndProc (HWND hWnd,
 UINT message,
 WPARAM wParam,
 LPARAM lParam)
{
  switch (message) {
    case MM_WOM_OPEN:
    case MM_WOM_CLOSE:
    case MM_WOM_DONE:
    case MM_WOM_OPEN:
    case MM_WOM_CLOSE:
    case MM_WIM_DATA:
      // pass down to line driver to handle
      tm_Process_Event(message,wParam,lParam);
      return (0L);
    case WM_DDE_ACK:
      ServerAcknowledge(hWnd, (HWND)wParam, lParam);
      return (0L);
    case WM_TIMER:
      // pass down to line driver to handle
      tm_Process_Event(message,wParam,lParam);
      return (0L);
    case WM_DDE_ADVISE:
      ServerAcknowledge(hWnd, (HWND)wParam, 0L);
      /* simulates NACK */
return ( 0L );

case WM_DDE_POKE: /* client sending command to server */
    ServerPoke ( hwnd, ( HWND ) wParam, lParam );
    return ( 0L );

case WM_DDE_TERMINATE:
    ServerTerminate ( hwnd, ( HWND ) wParam );
    return ( 0L );

case WM_DDE_UNADVISE:
    ServerAcknowledge ( hwnd, ( HWND ) wParam, 0L );
    return ( 0L );  /* simulates NACK */

case WM_DDE_REQUEST:
    ServerAcknowledge ( hwnd, ( HWND ) wParam, 0L );
    return ( 0L );  /* simulates NACK */

case WM_DDE_EXECUTE:
    ServerAcknowledge ( hwnd, ( HWND ) wParam, 0L );
    return ( 0L );  /* simulates NACK */

default:
    return (DefWindowProc(hwnd, message, wParam, lParam));
}

/* function DDEWndProc */

FUNCTION: AboutDlgProc(HWND, UINT, WPARAM, LPARAM)

PURPOSE: Processes messages for "About" dialog box

BOOL FAR PASCAL __export AboutDlgProc ( HWND hDlg,
UINT message,
WPARAM wParam,
LPARAM lParam )
{
    switch ( message )
    {
    case WM_INITDIALOG:
        return ( TRUE );

    case WM_COMMAND:
        if (( wParam == IDOK ) || ( wParam == IDCANCEL ) )
        {
            EndDialog ( hDlg, TRUE );
            return ( TRUE );
        }
        break;
    }
    return ( FALSE );
FUNCTION: MainWndProc(HWND, UINT, WPARAM, LPARAM)

PURPOSE: Processes messages for server

long FAR PASCAL __export MainWndProc (  
    HWND    hWnd,  
    UINT    message,  
    WPARAM wParam,  
    LPARAM lParam
)
{
    HDC       hDC;
    TEXTMETRIC tm;
    PAINTSTRUCT ps;

    switch ( message ) {
        case WM_WOM_OPEN:
        case WM_WOM_CLOSE:
        case WM_WOM_DONE:
        case WM_WIM_OPEN:
        case WM_WIM_CLOSE:
        case WM_WIM_DATA:
            // pass down to line driver to handle
            tm_Process_Event ( message, wParam, lParam );
            break;
        case WM_CREATE:
            hDC = GetDC ( hWnd );
            SelectObject ( hDC, GetStockObject (SYSTEM_FIXED_FONT) );
            GetTextMetrics ( hDC, &tm );
            nCharW = tm.tmAveCharWidth;
            nCharH = tm.tmHeight;
            ReleaseDC ( hWnd, hDC );
            rect.top = 3 * nCharH / 2;
            break;
        case WM_SIZE:
            rect.right = LOWORD ( lParam );
            rect.bottom = HIWORD ( lParam );
            UpdateWindow ( hWnd );
            break;
        case WM_SETFOCUS:
            SetFocus ( GetDlgItem ( hWnd, IDOK ) );
            break;
        case WM_PAINT:
            InvalidateRect ( hWnd, NULL, TRUE );
    }
} /* function AboutDlgProc */
hDC = BeginPaint ( hWnd, &ps );
/* do any painting here */

SetBkMode ( hDC, TRANSPARENT );
ShowTestMode ( );
EndPaint ( hWnd, &ps );
break;

case WM_COMMAND:
    switch ( wParam )
    {
        case IDM_ABOUT:
            DialogBox ( hWnd,
                MAKEINTRESOURCE(IDD_ABOUT),
                // this name must match the one
                "ABOUT", // used in the RC file
                hWnd,
                MakeProcInstance ( AboutDlgProc, hWnd ) );
            break;

        case ID_TAB:
        case ID_SHIFT_TAB:
            break;
    }
    break;

case WM_DDE_INITIATE:
    ServerInitiate ( ( HWND ) wParam, lParam );
    break;

case WM_DESTROY:
    break;

    case WM_SYSCOMMAND:
        if (!(( wParam == SC_CLOSE ) && ( !bShuttingDown )) ) {
            // tricky: don't let Windows process this message yet
            // we will shut down first, then tell Windows about it
            PostQuitMessage ( 0 ); // to pop us out of main
            break;
        } else
        return ( DefWindowProc ( hWnd, message, wParam, lParam ) );

    case WM_TIMER:
        if ( wParam == WINMAIN_TIMER_ID ) {
            // time to do polling
            do_polling();
        } else {
            // pass down to line driver to handle
            tm_Process_Event ( message, wParam, lParam );
        }
        break;

    default:
        return ( DefWindowProc(hWnd, message, wParam, lParam) );
    return ( 0L );
FUNCTION: InitApplication(HANDLE)

PURPOSE: Initializes window data and registers window class

/******************************************************************************/

static

BOOL InitApplication ( )
{
    WNDCLASS      wc;

    wc.style = 0;
    wc.lpfnWndProc = MainWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    // wc.hIcon = LoadIcon((HINSTANCE)NULL, IDI_APPLICATION);
    // wc.hIcon = LoadIcon ( hInstance, MAKEINTRESOURCE (IDI_ICON1));
    wc.hCursor = LoadCursor ( ( HINSTANCE ) NULL, IDC_ARROW );
    wc.hbrBackground = COLOR_WINDOW + 1;
    wc.lpszMenuName = "ServerMenu"; // this name must match the
                                    // one used in the RC file
    wc.lpszClassName = "voysServerWClass";

    if ( !RegisterClass ( &wc ) )
        return ( FALSE );

    wc.style = 0;
    wc.lpfnWndProc = DDEWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = ( HICON ) NULL;
    wc.hCursor = ( HCURSOR ) NULL;
    wc.hbrBackground = 0;
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "voysServerDDEWndClass";

    return ( RegisterClass ( &wc ) );

} /* function InitApplication */

/******************************************************************************/

FUNCTION: InitInstance(HANDLE, int)

PURPOSE: Saves instance handle, creates main window, and
        creates 3 child edit controls with id's 1, 2, and 3.

/******************************************************************************/

static
BOOL InitInstance (
    const HANDLE hInstance,
    const int nCmdShow
)
{
    HDC hDC;
    TEXTMETRIC tm;
    int nHorzRes, nVertRes;
    int xDelta, yDelta;

    InitAckTimeOut ( );   /* in module SERVDDE */

    hInst = hInstance;

    hWndMain = CreateWindow ( "woysServerWClass", 
                              SERVER_TITLE, 
                              WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, 
                              CW_USEDEFAULT, 
                              CW_USEDEFAULT, 
                              CW_USEDEFAULT, 
                              ( HWND ) NULL, 
                              ( HMENU ) NULL,
                              hInstance, 
                              NULL
    );

    if ( !hWndMain )
        return ( FALSE );

    hDC = GetDC ( hWndMain );
    hWndDisplay = hDC;
    GetTextMetrics ( hDC, ( LPTEXTMETRIC ) & tm );
    xDelta = tm.tmAveCharWidth;
    yDelta = tm.tmHeight + tm.tmExternalLeading;
    nHorzRes = GetDeviceCaps ( hDC, HORZRES );
    nVertRes = GetDeviceCaps ( hDC, VERTRES );
    ReleaseDC ( hWndMain, hDC );

    MoveWindow ( hWndMain, 
                 nHorzRes / 2 + xDelta * 6, 
                 nVertRes / 2 + yDelta, 
                 xDelta * 35, 
                 yDelta * 12, 
                 FALSE );

    // ShowWindow(hWndMain, nCmdShow);
    // ShowWindow(hWndMain, SW_SHOWNMINIMIZED);
    ShowWindow ( hWndMain, SW_SHOWNNOACTIVE );
    UpdateWindow ( hWndMain );

    return ( TRUE );
}

/* function InitInstance */
static
int app_startup(
    const char *cmdline,
    const HINSTANCE hInstance,
    const HWND hWnd
)
{
    int nRetVal;    // Voysys return value
    int nLine;
    int dir_num;
    DWORD dwVersion;

    Debug_flag = 3;

    nRetVal = parse_arguments(cmdline);
    if (nRetVal != RC_SUCCESS)
        goto done;

    nRetVal = CheckDebugSanity();
    if (nRetVal != RC_SUCCESS)
    {
        Debug (1, sprintf(Debug_buf, "IPC WD_S/app_startup: 
                           debug sanity failure") );
        goto done;
    }

    dwVersion = GetVersion();    /* get Windows, DOS versions */

    nRetVal = tm_Initialize(
        ld_hw_irq,
        ld_sw_int,
        hInstance,
        hWnd,
        promptext,
        ifileext
    );

    if (nRetVal != RC_SUCCESS)
        goto done;

    /* initialize array of line states */
    for (nLine = 0; nLine < MAX_AL; nLine++)
    {
        line[nLine].bConnected = FALSE;
        line[nLine].bCmdInProgress = FALSE;
        line[nLine].bTestMode = FALSE;

        /* initialize voice file directory names */
        for (dir_num = 0; dir_num < MAX_DIRS; dir_num++)
        {
            line[nLine].dirname[dir_num][0] = '\0';
        }
    }

    next_user_filename = 0;
    nRetVal = RC_SUCCESS;

    done:
    return nRetVal;
}

 />

function app_startup */
static
int app_shutdown ( void )
{
    int nRetVal;       // Voysys return value
    int nLineNum;
    struct command cmd;  // command to execute
    struct event   event;  // event received
    int count;
    boolean gotone;
    MSG msg;
    HANDLE hAccel;
    boolean bSuccess;

    bShuttingDown = TRUE;

    nRetVal = tm_Shutdown();
    if (nRetVal != RC_BUSYLINE)
        goto finish;

    // some line is connected or has a command in progress
    // give it some time to abort or disconnect
    count = PEEK_MSG_COUNT;
    while (count--) {
        gotone = PeekMessage (&msg, (HWND)NULL, 0, 0,
                                PM_REMOVE);

        if (gotone) {
            if (!TranslateAccelerator (hWndMain, hAccel, &msg)) {
                TranslateMessage (&msg);
                DispatchMessage (&msg);
            }
        }
        do_polling();
    }

    nRetVal = tm_Shutdown();
    if (nRetVal != RC_BUSYLINE)
        goto finish;

    // for any lines connected but not in progress, inject a
    disconnect command
    for (nLineNum = 0; nLineNum < MAX_AL; nLineNum++) {
        lp = &line[nLineNum];
        if (lp->bConnected) {
            cmd.nCommand = DVRC_DISCONNECT;
            cmd.nLine = nLineNum;
            lp->bCmdInProgress = TRUE;
            nRetVal = tm_Do_Command ( cmd,
                                      TRUE, // start of command
                                      &event
                                    );
            if (nRetVal != RC_CMDNOTDONE) {
                /* send event back to client */
                send_event ( event,
                             &nRetVal
                           );
            }
        }
    }

    finish:;
}
lp->bCmdInProgress = FALSE;

// give commands time to work
count = PeekMessage ( &msg, ( HWND ) NULL, 0, 0,

PM_REMOVE);

if ( gotone ) {
    if ( !hDlgCurrent || !IsDialogMessage ( hDlgCurrent,

    &msg ) ) {
        if ( !TranslateAccelerator ( hWndMain, hAccel, &msg

        ) ) {
            TranslateMessage ( &msg );
            DispatchMessage ( &msg );
        }
    }
}

do_polling();

// try one more time
// if it doesn't work, give up
nRetVal = tm_Shutdown();

finish:

/* Terminate all DDE conversations before destroying
client window */
TerminateConversations( );

// now tell Windows to close our windows
bSuccess = PostMessage ( hWndMain,

WM_SYSCOMMAND,

SC_CLOSE,

0);

if (!bSuccess) {
    Debug ( 1, sprintf ( Debug_buf, "IPC_WD_S/app_shutdown:

    PostMessage failed" ) );
}

// chew up messages that close windows
count = PeekMSG_COUNT;
while ( count-- ) {
    gotone = PeekMessage ( &msg, ( HWND ) NULL, 0, 0,

    PM_REMOVE);

    if ( gotone ) {
        if ( !hDlgCurrent || !IsDialogMessage ( hDlgCurrent,

        &msg ) ) {
            if ( !TranslateAccelerator ( hWndMain, hAccel, &msg

            ) ) {
                TranslateMessage ( &msg );
                DispatchMessage ( &msg );
            }
        }
    }
}

Debug ( 5, sprintf (Debug_buf, "IPC_WD_S/app_shutdown: ret"));
return RC_SUCCEED;
}

FUNCTION: WinMain(HANDLE, HANDLE, LPSTR, int)

PURPOSE: Calls initialization function, processes message loop

******************************************************************************

int PASCAL WinMain {
HANDLE hInstance,
HANDLE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow
}

MSG msg;
HANDLE hAccel;
BOOL done;
BOOL gotone;
int nRetVal; // Voysys return value
int retval;
UINT timerID;
BOOL bAnyError;

retval = 0;
bAnyError = FALSE;

if (!hPrevInstance) {
    /* delete, create and open debug file */
    debugfile = open ("vsdebug.txt",
        O_CREAT | O_TRUNC | O_TEXT | O_WRONLY,
        S_IWRITE
    );
    if (debugfile == -1)
        return (0); // error to open the debug file

    if (!InitApplication (hInstance)) {
        bAnyError = TRUE;
        goto stop3;
    }
} else {
    /* Another Server is running; abort */
    goto stop4;
}

if (!InitInstance (hInstance, nCmdShow)) {
    bAnyError = TRUE;
    goto stop3;
}

hAccel = LoadAccelerators(
hInstance,
    "ServerAcc" // must match name in RC file
);
nVRetVal = app_startup ( lpCmdLine, hInstance, hWndMain );
if ( nVRetVal != RC_SUCCESS ) {
    bAnyError = TRUE;
    goto stop3;
}

/* start polling timer */
timerID = SetTimer ( hWndMain, WINMAIN_TIMER_ID, 
                EV_POLL_DELAY, NULL );
if ( timerID != WINMAIN_TIMER_ID ) {
    bAnyError = TRUE;
    goto stop2;
}

done = FALSE;
while ( !done ) {
    gotone = PeekMessage ( &msg, ( HWND ) NULL, 0, 0, 
                PM_REMOVE );
    if ( gotone ) {
        if ( !hDlgCurrent || !IsDialogMessage ( hDlgCurrent, 
                    &msg ) ) {
            if ( TranslateAccelerator(hWndMain, hAccel, &msg) ) {
                TranslateMessage ( &msg );
                DispatchMessage ( &msg );
            }
            if ( msg.message == WM_QUIT )
                done = TRUE;
        }
    }
}
retval = msg.wParam;

//stop1:
/* remove timer */
KillTimer ( hWndMain, timerID );

stop2:
app_shutdown ( );

stop3:
close ( debugfile );
if ( bAnyError )
    MessageBox( hWndMain, "Server cannot be started.\nPlease see output debug file for more error message.", 
            SERVER_TITLE, MB_ICONSTOP | MB_OK );

stop4:
    return ( retval );

} /* function WinMain */
APPENDIX G
LD_Dial.c
Copyright 1992-1994 Vovsys Corporation

5
	//*****************************************************************************/
	/* Name: LD_Dial.c */
	/* Description: */
	/* Line Driver layer for 4-channel Dialogic line card. */
	/* Accepts commands from InterProcess Communication layer and */
	/* executes them on the Dialogic lines. */
	/* */
	/* Supports Dialogic line card models D4x (D4IB, D41D, D21D */
	/* tested so far). */
	/* */
	/* $Log: /VoysAccess/core/1d/1D_DIAL.C */
	/* */
	//*****************************************************************************/

#include <ctype.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include <io.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// stuff from windows.h; needed for ld_Process_Event

// can't just include the file; would have conflict defining
struct DCB { // Dialogic Control Block
typedef unsigned long DWORD;
typedef unsigned int UINT;
typedef signed long LONGLONG;
typedef UINT WPARAM;
typedef LONGLONG LPARAM;
#define WM_TIMER 0x0113
#define PROMPT_FEXT "VOX"
#define INDEX_FEXT "VOI"

#include "d40.h"
#include "d40lib.h"
#include "vfcns.h"

extern int _int_level; /* software interrupt level */

#include "OS.h"
#include "dVR.h"

#include "Command.h"
#include "LD.h"
#include "VR.h"

01
#define DO_SETXPM 0

// after determining real lines, fill up array with fake lines?
#define DO_FAKE_LINES 0

/******************************************************************************
/* we number lines from 0 thru N-1; Dialogic numbers channels 1 thru N */

#define CHANNEL(LINE) ((LINE)+1)
#define LINE(CHANNEL) ((CHANNEL)-1)

/* actual number of lines present */
static
unsigned int nActualLines = 0;

/******************************************************************************/

/* in-memory struct to access an indexed prompt file */
struct ifmap {
    int ifhandle;   /* handle to indexed file */
    struct {
        char fname[MAX_FN];   /* base file name */
        long start;  /* starting byte in file */
        long length; /* length in bytes */
    } indextab[MAX_PHRASE];
    int num_phrases; /* number of phrases in indextab */
};

/* struct that specifies a play-list, to xplayf */
struct findex {
    int type;     /* type of i/o transfer */
    int FileHandle;   /* DOS file handle */
    long BlockPosition; /* offset of block relative to start of file */
    long BlockLength; /* length of block */
};

/******************************************************************************

#define SET_TIMEOUT_TO(N) lp->timeout = (N)

// Timeout for operations that should never fail.
// Actually, timeout for events that should come almost
// immediately; if they don't, something is broken.
#define TIMEOUT_FOR_BROKEN (time_in_msec() + (10*1000))

// Timeout for N seconds from now.
#define TIMEOUT_FOR_SEC(N) (time_in_msec() + ((N)*1000))

// Timeout after going on-hook.
// Makes sure that CO sees that we are on-hook.
// If we didn't wait, there is a chance that we would immediately
// try to initiate another call, and CO might see such a quick
// on-and-then-off-hook that it looks like a wink or something.
#define TIMEOUT_AFTER_ONHOOK (time_in_msec() + (3*1000))

/******************************************************************************/

/ *
* Information about state of each active line.
* /
struct line_t {
  boolean bExists; /* is this entry used for a line ? */
  int nLineNum;  /* line number (0 to MAX_AL-1) */
  boolean bConnected; /* is a client connected to line? */
  boolean bCmdInProgress; /* is there a cmd in progress? */
  int nCmdStage; /* current step of command being executed */
}

struct command cmd; /* command that is in progress */
struct event event; /* event being constructed */
clock_t timeout; /* if != 0, time to abort cmd */
int nMaxDTHMF; /* end play or record if this many digits received */
int nTempVRetVal; /* ultimate value of ep->result */
boolean bSendEvent; /* event filled in; ready to return*/
int num_events; /* number of queued events for line */

struct {
  boolean bIsWinEvent; /* is this a Windows event ? */
  // fields for Windows event
  DWORD dwMsg;
  DWORD dwParam1;
  DWORD dwParam2;
  DWORD dwDevice;
  // fields for Dialogic event
  int nDialEvent; /* Dialogic event number */
  int nDialCallState; /* Dialogic call state number */
} queued_ev[MAX_EVENTS_PER_LINE];

#define IS_AN_EVENT        ((lp->num_events > 0)
#define NEXT_EVENT         ((lp->queued_ev[0])
#define TIMEOUT_EVENT      ((lp->queued_ev[0].bIsWinEvent) & (lp->queued_ev[0].dwMsg == WM_TIMER))
#define REMOVE_EVENT       (int i; 
                          for(i=0; i<(lp->num_events)-1; i++) 
                          lp->queued_ev[i] = lp->queued_ev[i+1]; 
                          lp->num_events--; )
#define REMOVE_ALL_EVENTS  (lp->num_events = 0)
#define EVENTQ_FULL        (lp->num_events == MAX_EVENTS_PER_LINE)
#define ADD_EVENT(MSG,P1,P2,DEV)   (lp->queued_ev[lp->num_events].bIsWinEvent = TRUE; 
                                        lp->queued_ev[lp->num_events].dwMsg = (MSG); 
                                        lp->queued_ev[lp->num_events].dwParam1 = (P1); 
                                        lp->queued_ev[lp->num_events].dwParam2 = (P2); 
                                        lp->num_events++; )
#define ADD_DIALOOGIC_EVENT(EV,CS)  (lp->queued_ev[lp->num_events].bIsWinEvent = FALSE; 
                                        lp->queued_ev[lp->num_events].nDialEvent = (EV); 
                                        lp->queued_ev[lp->num_events].nDialCallState = (CS); 
                                        lp->num_events++; )

boolean bUserIsOffHook; /* is user/caller off-hook ? */
boolean bCardIsOffHook; /* card has line off-hook ? */
boolean silent; /* for callout, is person silent ? */
#if defined(_WIN)
/* Windows */
#endif
// The following pointer variables should be initialized to
// all zeros.
long rwb_DosPointer;
short rwb_ProtectedSelector;
long rwb_ProTECTEDPointer;
long xrwD_DosPointer;
short xrwD_PROTECTEDIndex; /* for play, for playing
long xrwD_PROTECTEDPointer; long fi_DosPointer;
short fi_PROTECTEDSelector; long fi_PROTECTEDPointer;
long dtmf_DosPointer;
short dtmf_PROTECTEDSelector; long dtmf_PROTECTEDPointer;
long cpb_DosPointer;
short cpb_PROTECTEDSelector; long cpb_PROTECTEDPointer;
long csb_DosPointer;
short csb_PROTECTEDSelector; long csb_PROTECTEDPointer;
#endif
int open_file;  /* for play/rec, voice file handle */
struct findex Files_Index[MAX_PF]; /* for play, for playing
boolean close_after[MAX_PF]; /* close file after play
boolean voice_has_ended;  /* for play/rec, voice data done?
boolean digits_have_ended; /* for play/rec, DTMF string
done ? */
struct ifmap ifile[MAX_IFILE]; /* for play, indexed voice
int num_open_file;  /* for play, num open indexed files */
/* stuff set from initplay, addplay, play commands */
struct pitem plist[MAX_PF];  /* play list items */
int plist_count;  /* num files in play list */
};
static
struct line_t line[MAX_AL] = {0};
/* variables used by all functions in this file */
/* they are set to point to current line[]. */
static struct line_t *lp = 0;  /* ptr to "line[aline_num]"
static struct command *cp = 0;  /* ptr to "line[aline_num].cmd"
static struct event *ep = 0;  /* ptr to "line[aline_num].event"
/* buffers used for DTMF input */
/* must be in global (can't be static ?) */
/* must not be inside a structure */
/* must be compiling program in large memory model */
char dial_dtmfbuf[MAX_AL] [MAX_DIGITS + 1] = {0};
/* variables for Check_For_Event */
static boolean Initialized = FALSE;  /* ld_initialize called ? */
#ifdef(R_W)  /* Windows */
/* variables for Check_For_Event */
long cfe_DosPointer = OL; short cfe_PROTECTEDSelector = 0;
long cfe_PROTECTEDPointer = OL;
char cDosMemErr[] = " failed to allocate DOS memory ";
#endif

#define CERM_STANDARD (C_LC + C_LCON + C_OFFH + C_ONH)

#define LS_WH_ON_HOOK 0x08 /* we have the line off-hook */
#define LS_NO_RINGING 0x10 /* no in-bound ring detected */
#define LS_LOOP_CURR_ON 0x20 /* caller has the line on-hook */
#define LS_DTMF_DETECTED 0x40 /* DTMF detected */
#define LS_SILENCE_DETECTED 0x80 /* silence detected */

#define LTERM "T_NOTERM",
#define LTERM "T_MAXDT",
#define LTERM "T_TERM",
#define LTERM "T_STOP",
#define LTERM "T_DOERR",
#define LTERM "T_MAXBYT",
#define LTERM "T_HFAIL",
#define LTERM "T_TIME",
#define LTERM "T_OFFH",
#define LTERM "T_DIAL",
#define LTERM "T_SIL",
#define LTERM "T_TOP",
#define LTERM "T_LCTERM",
#define LTERM "T_DFULL",
#define LTERM "T_ONH",
#define LTERM "15",
#define LTERM "16",
#define LTERM "T_MTERM",
#define LTERM "T_CATERM",
#define LTERM "T_LCREV",
#define LTERM "T_LC",
#define LTERM "T_RING",
#define LTERM "T_SILOFF",
#define LTERM "T_SILON",
#define LTERM "T_AMXCON",
#define LTERM "T_AMXDIS",
#define LTERM "T_LCON",
#define LTERM "T_MAXERG",
#define LTERM "T_MCTERM",
#define LTERM "T_MDTMF",
#define LTERM "T_IDTIME",
#define LTERM "T_NSIL",
#define LTERM "T_BUFFUL",
#define LTERM "T_BUFFRM",
#define LTERM "T_EMSERR",
#define LTERM "T_EMISLOW",
#define LTERM "T_EMISHI",
#define LTERM "T_STPARM",
#define LTERM "T_WINK",
#define LTERM "T_WKRRCV",
"T_DTMP",
"T_TONEON",
"T_TONEOFF",
"T_BADTERM",
"T_MFONE",
"T_MFONEOFF",
"T_CAERROR",
"T_TGERR",
"T_TGCMPLT",
"T_ADSIERR"
};

/**
 * Find_Phrase_In_IFile -- See if phrase can be obtained from
 * an indexed file. Convert the play item if possible.
 */
static void Find_Phrase_In_IFile ( 
  struct pitem * playitem /* play item to convert */
) {
  int ifile_num; /* indexed file number (0 to MAX_IFILE-1) */
  int pcount; /* phrase count */
  char basename[MAX_FN]; /* base file name */
  /* extract base filename from full filename */
  fname_full_to_base ( 
    playitem->fname, /* input -- full filename */
    basename /* retval -- base name */
  );
  playitem->ifnum = ( -1 ); /* not found in ifile */
  for ( ifile_num = 0; ifile_num < lp->num_openifile; 
    ifile_num++ ) {
    for ( pcount = 0; pcount < 
      lp->ifile[ifile_num].num_phrases; pcount++ ) {
      if ( strcmp ( basename, 
        lp->ifile[ifile_num].indextab[pcount].fname
      ) == 0 ) { /* found prompt in indexed file */
        playitem->ifnum = ifile_num;
        playitem->index = pcount;
        return;
      }
    }
  } /* didn't find prompt in indexed file */
}

/**
 * Get_Line_Status -- See if current line is on or off hook,
 * etc.
 */


static int Get_Line_Status ( CSB *lpCSB )
{
    CSB My_csrb;    /* Channel status block */
    int nDialRetVal; // Dialogic return value
    int nVRetVal = 0;

    /* Get Channel Status */

    #if defined(E_W)    /* Windows */
    alloc_DOS_mem ( sizeof ( My_csrb ),
        & ( lp->csb_DosPointer ),
        & ( lp->csb_ProtectedSelector ),
        & ( lp->csb_ProtectedPointer ) );
    if ( lp->csb_ProtectedSelector == 0 ) {
        nVRetVal = RC_RAMPULL;
        goto done;
    }
    #endif

    nDialRetVal = getcstat ( /* Get Channel Status */
        CHANNEL ( lp->nLineNum ),
    #if defined(E_W)    /* CSB */ ( lp->csb_DosPointer ) /* Windows */
    #else
        &My_csrb
    #endif
    );

    #if defined(E_W)    /* Windows */
    memcpy ( &My_csrb, ( void * ) lp->csb_ProtectedPointer,
        sizeof ( My_csrb ) );
    free_DOS_mem ( & ( lp->csb_DosPointer ),
        & ( lp->csb_ProtectedSelector ),
        & ( lp->csb_ProtectedPointer ) );
    #endif

    switch ( nDialRetVal ) {
        case 0:        /* request complete */
            break;
        case 4:        /* system not initialized */
            case 9:        /* invalid channel number */
                nVRetVal = RC_BADLINE;
                break;
        default:        /* some unknown error */
                nVRetVal = RC_BADLINE;
                break;
    }

    done:
    *lpCSB = My_csrb;
    return ( nVRetVal );
}

 /************************************************************************************
 *  Flush_All_DTMF -- flush all DTMF on current line.  
 ************************************************************************************/

static void Flush_All_DTMF ( void )
int nDialRetVal; // Dialogic return value

/ *
 * Flush all DTMF on this channel.
 */
nDialRetVal = clrdtmf (CHANNEL (lp->nLineNum));

switch (nDialRetVal) {
  case E_SUCC:    /* Function was successful. */
  case E_NODT:   /* Dtmf buffer empty. */
    /*success*/
    lp->bSendEvent = FALSE;
    break;
  case E_BADDL:   /* D4x hardware error. */
    ep->result = RC_BADLINE;
    lp->bSendEvent = TRUE;
    break;
  default:        /* some unknown error */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    break;
}

/* **************************************************************************
 * Set_Channel_Mask    -- set event mask on a line.
 **************************************************************************/

static void Set_Channel_Mask (int mask, /* event mask (sum of CEM's) */
  int rings /* number of rings for inbound call */
) 
{
  int nDialRetVal; // Dialogic return value

  nDialRetVal = setcat (CHANNEL (lp->nLineNum),
                        mask,
                        rings);

  switch (nDialRetVal) {
    case E_SUCC: /* Function was successful. */
    /*success*/
    lp->bSendEvent = FALSE;
    break;
    case E_BADDL: /* D4x hardware error. */
    ep->result = RC_BADLINE;
    lp->bSendEvent = TRUE;
    break;
    default: /* some unknown error */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    break;
  }
}
/***********************
* Check_User_Connection  --  check if user is still on-line
***********************/

static
int  Check_User_Connection()
{

    CSB    My_csb;
    int    nLineStatus;
    int    nDialRetVal;  // Dialogic return value

    nDialRetVal = Get_Line_Status( &My_csb );
    if ( nDialRetVal ) return( nDialRetVal );

    nLineStatus = My_csb.linetstat;
    if ( nLineStatus & LS_LOOP ) {
        Debug ( 3, sprintf ( Debug_buf, "%d:
            LD_Dial/Check_User_Connection: user on hook
            (linetstat = %x)", nLineStatus )
        );
        lp->bUserIsOffHook = FALSE;
        nDialRetVal = RC_USERONHOOK;
    }
    if ( nLineStatus & LS_HOOK ) {
        Debug ( 3, sprintf ( Debug_buf,
            "%d:  LD_Dial/Check_User_Connection: card on hook
            (linetstat = %x)", nLineStatus )
        );
        lp->bCardIsOffHook = FALSE;
        nDialRetVal = RC_CARDONHOOK;
    }

    return( nDialRetVal );
}

/******************************/
*  Macros that only work inside the Do*Command functions
******************************/

#define IF_NO_EVENT_THEN_BREAK  
    if (!(IS_AN_EVENT) )
        break;

#define IF_TIMEOUT_THEN_INTERNALFAIL
    if (TIMEOUT_EVENT) {
        REMOVE_EVENT;
        ep->result = RC_INTERNALLINE;
        lp->bSendEvent = TRUE;
        goto done;
    }

#define IF_WRONG_EVENT_THEN_INTERNALFAIL(EV)
    if (NEXT_EVENT.dWMsg != (EV)) {
        REMOVE_EVENT;
        ep->result = RC_INTERNALLINE;
    }
- 230 -

    lp->bSendEvent = TRUE;    \ goto done;

5  #define IF_WINDOWS_EVENT_THEN_INTERNALFAIL
    if (NEXT EVENT.bIsWinEvent) {\ REMOVE EVENT;
    ep->result = RC_INTERNALFAIL;
    lp->bSendEvent = TRUE;
    goto done;
}

10 #define REMOVE_TIMEOUTEVENTS {while (TRUE) {
    if (!TIMEOUT_EVENT) break;
    REMOVE EVENT; }

15 // stage value that means we want to abort the command
20 #define ABORT_STAGE 98
25 // stage value that means we want to abort the command because of
26 // server shutdown
27 #define SHUTDOWN_STAGE 99

30 /*---------------------------------------------*/
35 /* Check_For_Event - see if driver has an event for any line.*/
30 /*---------------------------------------------*/

35 static void Check_For_Event {
    boolean *GotEvent

35     int nDialRetVal;    // Dialogic return value
35     int nLineNum;      /* line number (0 to MAX_AL-1) */
35     int channel;
35     int nDialEvent;    // Dialogic event number
35     int nDialCallState;     // Dialogic call state number
35     char info[80];     /* info message */
35     EVTBLK My_event;    /* event block */
35     boolean got_vr_event;

40    *GotEvent = FALSE;

45    nDialCallState = 0;

50    /* try to get an event */
55    if defined(E_W)
55        if ( cfe_DosPointer == 0L ) {
55          Debug ( 1, sprintf ( Debug_buf, "LD_Dial/Check_For_Event:
55                                      cfe_DosPointer corrupted" ) );
55          goto done;
55        }
55    #endif
55    nDialRetVal = gtevtblk ( 55
55        nDialRetVal = gtevtblk ( 55
55        #if defined(E_W)
55          ( EVTBLK * ) cfe_DosPointer
55        #else
55          &My_event
55        #endif

60    #else
if defined(E_W)
    memcpy ( &My_event, ( void * ) cfe_ProtectedPointer,
             sizeof ( My_event ) );
#endif

if ( nDialRetVal > 0 ) {
    /* "get event" function failed ! */
    Debug ( 1, sprintf ( Debug_buf, "LD_Dial/Check_For_Event:
                     gtevtblk() ret %d", nDialRetVal ) );
    goto done;
}

/* if we didn't get a line event, check for a voice
   recognition event */
if ( nDialRetVal == 0 ) {
    vr_Check_For_Event ( &got_vr_event,
                         &LineNum,
                         &DialEvent,
                         &DialCallState
                     );
}

/* if we didn't get an event, return */
if ( ( nDialRetVal == 0 ) & & ( !got_vr_event ) )
    goto done;

/* we got an event, handle it */
if ( nDialRetVal != 0 ) {
    nLineNum = LINE ( My_event.devchan );
    nDialEvent = My_event.evrcode;
    nDialCallState = My_event.evtdata;
    /* no else clause; already put info in
       variables */

    /* if event for disallowed line, ignore */
    if ( nLineNum >= MAX_AL )
        goto done;

    lp = &line[nLineNum];
    if ( Debug_flag > 2 ) {
        if ( ( nDialEvent == T_SILOFF )
            || ( nDialEvent == T_SILON )
            || ( nDialEvent == T_LCON )
            || ( nDialEvent == T_LC )
            || ( nDialEvent == T_DTMF )
            || ( nDialEvent == T_DOSERR )
            || ( nDialEvent == T_CATERM ) )
            sprintf ( info,
                      "received Dialogic event %d ('%s') callstate %d",
                      nDialEvent,
                      DialEventNumToName[nDialEvent],
                      nDialCallState
                  );
        else
            sprintf ( info,
                      "received Dialogic event %d ('%s')",
                      nDialEvent,
                      );
    }
DialEventNumToName[nDialEvent];
}
if ( nLineNum >= ( int ) nActualLines ) {
    if ( Debug_flag > 0 ) {
        printf (
            "received Dialogic event \%d from out-of-range channel \%d",
            nDialEvent, channel
        );

    }
    goto done;
}
/* check for illegal event number */
else if ((nDialEvent < T_VR_FIRST) || (nDialEvent > T_VR_LAST))
    && ((nDialEvent < T_NOTERM) || (nDialEvent > MAXTERM)) {
    if ( Debug_flag > 0 ) {
        printf (
            "received out-of-range Dialogic event \%d",
            nDialEvent
        );

    }
    goto done;
} else {
    /* events that tell whether caller is on- or off-hook; don't pass in */
    if ( nDialEvent == T_LC ) {
        /* call_state gives duration in 10 ms ticks */
        /* on-hook for more than 4.8 seconds ? */
        if ( nDialCallState > 480 )
            lp->bUserIsOffHook = FALSE;
    } else if ( nDialEvent == T_LCON )
        /* do nothing */
    else {
        if ( nDialEvent == T_ONH )
            lp->bCardIsOffHook = FALSE;
        /* put info into line[] array */
        ADD_DIALOGIC_EVENT(nDialEvent,nDialCallState)
        *GotEvent = TRUE;
        goto done;
    }
    goto done;
}
done:
return;

/**************************************************************************
* Set_End_Conditions -- Check and set up end conditions for
* play or record, according to spec in command.
**************************************************************************/
static
void Set_End_Conditions (
    void
)
{
    char cstr[2];

    if ( (cp->end_voice_on_any_digit && cp->end_voice_on_digit_end )
        || (cp->end_oper_on_any_digit && cp->end_oper_on_digit_end ) )
    {
        Debug ( 4, sprintf ( Debug_buf, 
"ID.Dial/Set_End_Conditions: bad digit mixture" ) );
        ep->result = RC_BADCOMMAND;
        lp->bSendEvent = TRUE;
        goto done;
    }

    if ( cp->max_digits < 0 )
    {
        ep->result = RC_BADMAXDIGS;
        lp->bSendEvent = TRUE;
        goto done;
    }

    if ( cp->end_voice_on_digit_end )
    {
        /* if end_digit is not valid, error */
        cstr[0] = cp->end_digit;
        cstr[1] = '\0';
        if ( strpbrk ( "#0123456789", (char*)cstr ) == NULL )
        {
            ep->result = RC_BADCOMMAND;
            lp->bSendEvent = TRUE;
            goto done;
        }
    }

    /* tricky: 0=unlimited(MAX_DIGITS); also handle too big */
    lp->nMaxDTMF =
        ( ( (cp->max_digits == 0) )
        || (cp->max_digits > MAX_DIGITS )
        ) ? MAX_DIGITS : cp->max_digits );

    done:
    return;
} /* function Set_End_Conditions */

/***************************************************************************/

* DoPlayCommand -- Play the playlist associated with the
* current line.

***************************************************************************/

static
void DoPlayCommand (void)
{
    int ifile_num;  // indexed file number (0 to MAX_IFILE-1)
    int pcount;    // phrase count
    RWB My_xrwb;   // read/write block
    RWB My_rwb;    // read/write block
    int item;
    long size;     // file size, for single-prompt file
struct findex far *fptr;
int nDialRetVal; // Dialogic return value
int nDialEvent; // Dialogic event number

switch (lp->nCmdStage) {
    case 0: // start of command
ep->nEvent = EV_PLAY;
ep->string[0] = '\0';
/*
 * Do error-checking.
*/
if (lp->plist_count == 0) {
    ep->result = RC_EMPTYLIST;
    lp->bSendEvent = TRUE;
goto done;
}
if (!lp->bCardIsOffHook) {
    ep->result = RC_CARDONHOOK;
    lp->bSendEvent = TRUE;
goto done;
}
if (!lp->bUserIsOffHook) {
    ep->result = RC_USERONHOOK;
    lp->bSendEvent = TRUE;
goto done;
}
/*

nDialRetVal = Check_User_Connection();
if (nDialRetVal) {
    ep->result = nDialRetVal;
    lp->bSendEvent = TRUE;
goto done;
}
*/
Set_End_Conditions ( );
if (!lp->bSendEvent )
goto done;
if (cp->flush_digits_at_start ) {
    Flush_All_DTMF ( );
    if (!lp->bSendEvent )
goto done;
}
/* Build play-list in lp->Files_Index[].
*/
for (item = 0; item < lp->plist_count; item++) {
    /* see if play-item is indexed file or
     single-prompt file */
    ifile_num = lp->plist[item].ifnum;
pcount = lp->plist[item].index;
    if (ifile_num >= 0) {
        /* indexed file */
        lp->Files_Index[item].type = 0;
        lp->Files_Index[item].FileHandle =
        lp->ifile[ifile_num].ifhandle;
        lp->Files_Index[item].BlockPosition =
        lp->ifile[ifile_num].indextab[pcount].start;
- 235 -

lp->Files_Index[item].BlockLength =
lp->ifile[ifile_num].indextab[pcount].length;
lp->close_after[item] = FALSE;
} else {
    /* single-prompt file */
    /* open the file */
    lp->Files_Index[item].FileHandle = open (  
        lp->plist[item].fname, O_BINARY | O_RDONLY );
    if ( lp->Files_Index[item].FileHandle < 0 ) {
        ep->result = RC_NOMFILE;
        ep->C_error = errno;
        ep->DOS_error = doserrno;
        Debug ( 3, sprintf ( Debug_buf,  
            "LDB Dial/DoPlayCommand: open failed %d",  
            lp->Files_Index[item].FileHandle ) );
    } /* close all files (except indexed files) in list */
    while ( --item >= 0 )
        if ( lp->close_after[item] )
            close (  
                lp->Files_Index[item].FileHandle );
        lp->bSendEvent = TRUE;
    goto done;
}

lp->Files_Index[item].type = 0;
lp->Files_Index[item].BlockPosition = 0;
size = lseek (  
    lp->Files_Index[item].FileHandle,  
    0L, SEEK_END );
if ( size < 0 ) {
    ep->result = RC_BADFILE;
    ep->C_error = errno;
    ep->DOS_error = doserrno;
} /* close all files (except indexed files) in list */
    while ( --item >= 0 )
        if ( lp->close_after[item] )
            close (  
                lp->Files_Index[item].FileHandle );
    lp->bSendEvent = TRUE;
    goto done;
}
}
lp->Files_Index[lp->plist_count].BlockPosition = (-1);
lp->Files_Index[lp->plist_count].BlockLength = (-1);

/*

* Build read/write Block.
*/
clrrwb ( &My_rw );
#if defined(R_W)  /* Windows */
alloca_DOS_mem (  
    (lp->plist_count + 1) * sizeof ( struct findex ),
    &lp->fiDireccion ),
    &lp->fi_ProtectedSelector ),
    &lp->fi_ProtectedSelector );
#endif

if ( lp->fi_ProtectedSelector == 0 ) {
    ep->result = RC_RAMFULL;
lp->bSendEvent = TRUE;
goto done;
}
memcpy {
    ( void * ) lp->fi.ProtectedPointer, /* dest */
    ( void * ) lp->Files_Index,
    ( void * ) lp->plist_count + 1
); /* source */

#else
fptr = ( void * ) ( lp->fi.DosPointer );
#endif

My_rwb.indexseg = FP_SEG ( fptra );
My_rwb.indexoff = FP_OFF ( fptra );

/* set parameters */
if ( cp->end_voice_on_any_digit )
    My_rwb.termdtmf = '"';
else if ( cp->end_voice_on_digit_end )
    My_rwb.termdtmf = cp->end_digit;
else
    My_rwb.termdtmf = '\0';
My_rwb.maxdtmf = lp->maxDTMF;
My_rwb.rwbflags = 0;
My_rwb.loopsig = 1; /* terminate on hangup */
#if defined(E_W)
    /* Windows */
    alloc_DOS_mem {
        sizeof ( My_rwb ),
        & ( lp->rwb.DosPointer ),
        & ( lp->rwb.ProtectedSelector ),
        & ( lp->rwb.ProtectedPointer )
    };
    if ( lp->rwb.ProtectedSelector == 0 ) {
        ep->result = RC_RAMFULL;
        lp->bSendEvent = TRUE;
        goto done;
    }
    memcpy { ( void * ) lp->rwb.ProtectedPointer,
        ( void * ) &My_rwb, sizeof ( My_rwb ));
#endif

/* Start playing play-list. */

n.DialRetVal = xplayf {
    CHANNEL ( lp->nLineNum ),
    ( PM_FILES | PM_NDX ),
#if defined(E_W)
    /* Windows */
    ( RWB * ) lp->rwb.DosPointer
#else
    /* non-Windows */
    &My_rwb
#endif
}

switch ( n.DialRetVal ) {
    case E_SUCCE: /* Function was successful. */
        break;
    case E_BADDL: /* D4x hardware error. */
        ep->result = RC_BADLINE;
        lp->bSendEvent = TRUE;
        goto done;
    case E_BADPAR: /* Bad parameter. */
        ep->result = RC_INTERNAL;
lp->bSendEvent = TRUE;
goto done;
default: /* some unknown error */
   ep->result = RC_INTERNALLINE;
lp->bSendEvent = TRUE;
goto done;
}
lp->voice_has Ended = FALSE;
lp->digits_have Ended = FALSE;
lp->nCmdStage = 1;
break;

case 1: /* we expect an end-of-playing event */
   IF_NO_EVENT THEN_BREAK;
   IF_WINDOWS_EVENT THEN_INTERNALLINE;
   nDialEvent = NEXT EVENT.nDialEvent;
   REMOVE EVENT;
   switch ( nDialEvent ) {
      case T_SILOFF: /* Silence off. */
      case T_SILON: /* Silence on. */
         goto done; /* ignore; remain in same state */
   }
/* close all files (except indexed files) in play-list */
for ( item = 0; item < lp->plist_count; item++ )
   if ( lp->cClose_after[item] )
      close ( lp->Files_Index[item].FileHandle );
#if defined(E_W)
   free_DOS_mem ( &(lp->fi_DosPointer),
      &(lp->fi_ProtectedSelector),
      &(lp->fi_ProtectedPointer) );
   free_DOS_mem ( &(lp->rwb_DosPointer),
      &(lp->rwb_ProtectedSelector),
      &(lp->rwb_ProtectedPointer) );
#endif
switch ( nDialEvent ) {
   case T_MAXDT: /* Maximum DTMF digits received. */
   case T_TERMDT: /* Terminating DTMF digit received. */
   case T_MDTMF: /* Terminated by masked DTMF digit */
      lp->nTempVRetVal = RC_ENDFLAG;
      break;
   case T_EOL: /* Eof reached on playback. */
      // Get Line Status();
      lp->nTempVRetVal = RC_SUCCESS;
      break;
   case T_LCTERM: /* Terminate by drop in loop signal */
      ep->result = RC_USERHUNGUP;
   // don't bother getting any digits they may have entered (?)
   lp->bSendEvent = TRUE;
   goto done;
   case T_STOP: /* Rec/play/getdtmf stopped. */
   case T_HFAIL: /* Hardware failure. */
      ep->result = RC_BADLINE;
      lp->bSendEvent = TRUE;
      goto done;
   default: /* some unknown event */
      ep->result = RC_INTERNALLINE;
      lp->bSendEvent = TRUE;
      goto done;
}
if (!cp->get_digits_at_end) {
    ep->result = lp->nTempVRetVal;
    lp->bSendEvent = TRUE;
    goto done;
}

// get digits
clrxxrwb(&My_xxrb);
/* set parameters */
if (cp->end_oper_on_any_digit)
    My_xxrb.termcmdf = 'O';
else if (cp->end_oper_on_digit_end)
    My_xxrb.termcmdf = cp->end_digit;
else
    My_xxrb.termcmdf = '\0';
/* tricky: 0==unlimited(MAX_DIGITS); also handle too big */
My_xxrb.maxdtmf = lp->nMaxDTMF;
My_xxrb.isxxrb = 1;
cp->total_timeout = SEC_TO_MSEC(1); /* don't wait */
My_xxrb.maxsec = (word) MSEC_TO_SEC(cp->total_timeout);
/*
 * maxsil == add up all silence between digits -- not
 * what we want
 */
/* My_xxrb.maxsil = MSEC_TO_SEC(cp->start_timeout); */
My_xxrb.intrdig = (byte) MSEC_TO_SEC(cp->interdigit_to);
My_xxrb.xxrbflags = 0;
My_xxrb.loopsig = 1; /* terminate on hangup */
#if defined(_WIN)
    /* Windows */
    alloc DOS_mem {
        MAX_DIGITS + 1,
        &lp->dtmf_DosPointer,
        &lp->dtmf_ProtectedSelector,
        &lp->dtmf_ProtectedPointer
    };
#endif
if (lp->dtmf_ProtectedSelector == 0) {
    ep->result = RC_RAMPULL;
    lp->bSendEvent = TRUE;
    goto done;
}
#else
    fptr = (void *) lp->dtmf_DosPointer;
#endif
My_xxrb.xferseg = PP_SEG(fptr);
My_xxrb.xferoff = PP_OFF(fptr);
My_xxrb.maxbyteh = 0;
My_xxrb.maxbytel = MAX_DIGITS;
#if defined(_WIN)
    /* Windows */
    alloc DOS_mem {
        sizeof(My_xxrb),
        &lp->xxrb_DosPointer,
        &lp->xxrb_ProtectedSelector,
        &lp->xxrb_ProtectedPointer
    };
#endif
if (lp->xxrb_ProtectedSelector == 0) {
    ep->result = RC_RAMPULL;
    lp->bSendEvent = TRUE;
- 239 -

goto done;
}

memcpy ( ( void * ) lp->xrwb_ProtectedPointer,
( void * ) My_xrwb, sizeof ( My_xrwb ) );
#endif

/* start recording data */

nDialRetVal = getdtmfs ( CHANNEL ( lp->nLineNum ),
#if defined(E_W)
( RWB * ) lp->xrwb_DosPointer /* Windows */
#else
&My_xrwb /* non-Windows */
#endif
);

switch ( nDialRetVal ) {
  case E_SUCCE: /* Function was successful. */
    break;
  default: /* some unknown error */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    goto done;
  }

  /* now wait until we get an event */
  lp->nCmdStage = 2;
  goto done;
}

switch ( nDialEvent ) {  /* we expect an end-of-input event */
  IF_NO_EVENT_THEN_BREAK;
  IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
  nDialEvent = NEXT_EVENT.nDialEvent;
  REMOVE_EVENT;
  switch ( nDialEvent ) {
    case T_SILOFF: /* Silence off. */
    case T_SILON: /* Silence on. */
      goto done; /* ignore; remain in same state */
  }
#endif

memcpy ( dial_dtmfbuf[lp - line], ( void * )
  lp->dtmf_ProtectedPointer,
  MAX_DIGITS + 1 );

free_DOS_mem ( &(lp->xrwb_DosPointer),
  &My_xrwb );

free_DOS_mem ( &(lp->dtmf_DosPointer),
  &My_xrwb );
#endif

switch ( T_MAXDT ) { /* Maximum DTMF digits received. */
  ep->result = RC_DATALENGTH;
  strcpy ( ep->string, dial_dtmfbuf[lp - line] );
  lp->bSendEvent = TRUE;
  goto done;
  case T_TERMDT: /* Terminating DTMF digit received. */
  case T_MDTMF: /* Terminated by masked DTMF digit*/
  ep->result = RC_ENDFLAG;
  strcpy ( ep->string, dial_dtmfbuf[lp - line] );
  lp->bSendEvent = TRUE;
  goto done;
  case T_TIMT: /* Rec/play/getdtmf timed out. */
  case T_SIL: /* Maximum silence received. */
case T_IDTIME: /* Interdigit delay exceeded. */
  strcpy ( ep->string, dial_dtmfbuf[lp - line] );
  if ( ep->string[0] == '\0' )
    ep->result = RC_STARTTIMOUT;
  else
    ep->result = RC_INERDIGITTIMEOUT;
  lp->bSendEvent = TRUE;
goto done;
case T_LCTERM: /* Terminate by drop in loop signal */
  ep->result = RC_USERHUNGUP;
  strcpy ( ep->string, dial_dtmfbuf[lp - line] );
  lp->bSendEvent = TRUE;
goto done;
  
case T_STOP: /* Rec/play/getdtmf stopped. */
  case T_HFAIL: /* Hardware failure. */
  ep->result = RC_BADLINE;
  lp->bSendEvent = TRUE;
goto done;
  
default: /* some unknown event */
  ep->result = RC_INERNALLINE;
  lp->bSendEvent = TRUE;
goto done;
}
/* never get here */
break;
case SHUTDOWN_STAGE: // server is shutting down
  ep->result = RC_SERVERSHUTDOWN;
  lp->bSendEvent = TRUE;
goto done;

default: // unknown command stage
  ep->result = RC_INERNA;
  lp->bSendEvent = TRUE;
goto done;
}
done:
// if complete, check to see if we have to do clean up
if (lp->bSendEvent) {
  // If exception happens while we are processing command,
  // free allocated DOS memory blocks. If the pointer
  // variables point to zeros, free DOS_mem() will just
  // return without doing anything.
  free_DOS_mem ( &(lp->fi_DosPointer),

  free_DOS_mem ( &(lp->fw_DosPointer),

  free_DOS_mem ( &(lp->xwb_DosPointer),

  free_DOS_mem ( &(lp->dtmf_DosPointer),

  return;
  // end function (DoPlayCommand)
void DoHangupCommand (void)
{
    int nDialRetVal;  // Dialogic return value
    int nDialEvent;  // Dialogic event number

    switch (lp->nCmdStage) {
        case 0: /* start of command */
            REMOVE_ALL_EVENTS;
            ep->nEvent = EV_HUNGUP;
            if (!lp->bCardIsOffHook) {
                ep->result = RC_CARDONHOOK;
                lp->bSendEvent = TRUE;
                goto done;
            }
            /* Ask channel to go on-hook. */
            nDialRetVal = sethook (CHANNEL (lp->nLineNum),
                                  H_ONH);

            switch (nDialRetVal) {
                case E_SUCCE: /* Function was successful. */
                    /* success */
                    break;
                case E_BADDL: /* D4x hardware error. */
                    ep->result = RC_BADLINE;
                    lp->bSendEvent = TRUE;
                    goto done;
                default: /* some unknown error */
                    ep->result = RC_INTERNALLINE;
                    lp->bSendEvent = TRUE;
                    goto done;

            }
            lp->nCmdStage = 1;
            /* now wait until we get an on-hook-complete event */
            goto done;

        case 1:
            IF_NO_EVENT THEN_BREAK;
            IF_WINDOWS_EVENT THEN_INTERNALFAIL;
            nDialEvent = NEXT_EVENT.nDialEvent;
            REMOVE_EVENT;
            switch (nDialEvent) {
                case T_SILOFF: /* Silence off. */
                case T_SILOFF: /* Silence on. */
                    goto done; /* ignore; remain in same state
                case T_ONH: /* Onhook complete. */
                    break;
                default: /* some unknown event */

            }
        }
    }

}
ep->result = RC_INTERNALLINE;
lp->bSendEvent = TRUE;
goto done;
}

// hung up successfully
// wait a few seconds to make sure CO sees we are hung up
lp->timeout = TIMEOUT_AFTER_ONHOOK;
REMOVE_TIMEOUT_EVENTS;
lp->nCmdStage = 2;
/* now wait until we get a timeout event */
goto done;

case 2:
    IF NO_EVENT THEN BREAK;
    if (!(TIMEOUT_EVENT) {
        REMOVE_EVENT;
        // ignore it and keep waiting
        goto done;
    }

    REMOVE_EVENT;
    /* make assumption */
    lp->bUserIsOffHook = FALSE;
    /* success */
    lp->bCardIsOffHook = FALSE;
    ep->result = RC_SUCCEED;
    lp->bSendEvent = TRUE;
    goto done;

case SHUTDOWN_STAGE: // server is shutting down
    ep->result = RC_SERVERSHUTDOWN;
    lp->bSendEvent = TRUE;
    goto done;

default: // unknown command stage
    ep->result = RC_INTERNAL;
    lp->bSendEvent = TRUE;
    goto done;
}

done:
    return;
} // end function (DoHangupCommand)

/**************************************************************************
* DoSendDtmfCommand -- Send DTMF digits out on the current line.
**************************************************************************/

static void DoSendDtmfCommand (void)
{  
  int nDialRetVal; // Dialogic return value
  char *ptr;
  int nDialEvent; // Dialogic event number
  #if defined(_WIN)
    long call_DosPointer = 0L;
  short call_ProtectedSelector = 0;
  long call_ProtectedPointer = 0L;
  #endif


switch (lp->nCmdStage) {
    case 0: /* start of command */
        ep->nEvent = EV_SENTDITMF;
        if ( !lp->bCardIsOffHook ) {
            ep->result = RC_CARDONHOOK;
            lp->bSendEvent = TRUE;
            goto done;
        }
        if ( !lp->bUserIsOffHook ) {
            ep->result = RC_USERONHOOK;
            lp->bSendEvent = TRUE;
            goto done;
        }
        /*
        nDialRetVal = Check_User_Connection();
        if ( nDialRetVal ) {
            ep->result = nDialRetVal;
            lp->bSendEvent = TRUE;
            goto done;
        }
        */
        /* Convert string to dial from Hayes format to Dialogic. */
        ptr = cp->string;
        while ( *ptr != '0' ) {
            if ( *ptr == 'A' )
                *ptr = 'a';
            else if ( *ptr == 'B' )
                *ptr = 'b';
            else if ( *ptr == 'C' )
                *ptr = 'c';
            else if ( *ptr == 'D' )
                *ptr = 'd';
            else if ( *ptr == '!' )
                *ptr = '&';
            ptr++;
        }
        /* Ask channel to send digits */
        #ifdef (WIN)
        /* Windows */
        alloc_DOS_mem ( /* Windows */
            strlen ( cp->string ) + 1,
            &call_DosPointer,
            &call_ProtectedSelector,
            &call_ProtectedPointer
        );
        if ( call_ProtectedSelector == 0 ) {
            ep->result = RC_RAMPULL;
            lp->bSendEvent = TRUE;
            goto done;
        }
        memcpy ( /* dest */
            ( void * ) call_ProtectedPointer,
            /* source */
            ( void * ) cp->string, /* source */
            strlen ( cp->string ) + 1
        );
        #endif
    }
nDialRetVal = dial ( CHANNEL ( lp->nLineNum ),
#if defined(_W)
   ( void * ) call_DosPointer
#else
   cp->string /* dial string */
#endif
);

#if defined(_W) /* Windows */
free_DOS_mem ( &call_DosPointer,
   &call_ProtectedSelector,
   &call_ProtectedPointer );
#endif

switch ( nDialRetVal ) {
   case E_SUCCEED: /* Function was successful. */
      /* success */
      break;
   case E_BADLINE: /* D4x hardware error. */
      ep->result = RC_BADLINE;
      lp->bSendEvent = TRUE;
      goto done;
   default: /* some unknown error */
      ep->result = RC_INTERNALLINE;
      lp->bSendEvent = TRUE;
      goto done;
}
lp->nCmdStage = 1;
/* now wait until we get an event */
goto done;

case EVT: /* we expect a digit-string-dialed event */
   IF_NOEVENT_THEN_BREAK;
   IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
   nDialEvent = NEXT_EVENT.nDialEvent;
   REMOVE_EVENT;
   switch ( nDialEvent ) {
      case T_SILENCE: /* Silence off. */
      case T_SILENCE: /* Silence on. */
      goto done; // ignore; remain in same state
      case T_DIAL: /* Dialing complete. */
          /* success */
          ep->result = RC_SUCCEED;
          lp->bSendEvent = TRUE;
          goto done;
      default: /* some unknown event */
          ep->result = RC_INTERNALLINE;
          lp->bSendEvent = TRUE;
          goto done;
   }
   /* never get here */
   break;

case SHUTDOWN_STAGE: // server is shutting down
   ep->result = RC_SERVERSHUTDOWN;
   lp->bSendEvent = TRUE;
   goto done;

default: // unknown command stage
   ep->result = RC_INTERNALLINE;
   lp->bSendEvent = TRUE;
   goto done;
done:
    // if complete, check to see if we have to do clean up
    if (lp->bSendEvent) {
        // do something here !!!
    }
    return;
} // end function (DoSendDTMFCommand)

******************************************************************************
*
* DoCallOutCommand -- Dial out on the current line.
*
******************************************************************************

static
void DoCallOutCommand (void)
{
    CPB My_cp;
    // Channel parameters
    CAR My_car;
    // Call Analysis Results
    int nDialRetVal;
    // Dialogic return value
    char *ptr;
    int nDialEvent;
    // Dialogic event number
    int nDialCallState;
    // Dialogic call state number

    #if defined(_W) // Windows */
    long call_DosPointer = 0L;
    short call_ProtectedSelector = 0;
    long call_ProtectedPointer = 0L;
    #endif

    switch (lp->nCmdStage) {
        case 0:  // start of command
            REMOVE_ALL_EVENTS;
            ep->nEvent = EV_CALLOUT;
            ep->voice_after_answer = 0;
            if ( lp->bCardIsOffHook ) {
                ep->result = RC_CARDOFFHOOK;
                lp->bSendEvent = TRUE;
            }
        goto done;
    }
    /* Convert string to dial from Hayes format to
    * Dialogic.
    * Note: Dialogic doesn't support "wait for dial-tone"
    */
    while ( *ptr != '\0' ) {
        if ( *ptr == 'A' )
            *ptr = 'a';
        else if ( *ptr == 'B' )
            *ptr = 'b';
        else if ( *ptr == 'C' )
            *ptr = 'c';
        else if ( *ptr == 'D')
            *ptr = 'd';
else if (*ptr == '!' )
    *ptr = '\';
/* turn "wait for dial-tone" into "pause" */
else if (*ptr == 'W' )
    *ptr = '\';
ptr++;
}

/* set channel parameters */
Set_Channel_Mask ( ( C_SILOFF + C_SILON + C_LC + C_LCON ), 1 );

if ( lp->bSendEvent )
goto done;
cclrpcb ( &My_cpb );
/* after this many RINGBACKs, give up */
My_cpb.nbrdna = cp->answer_rings;
My_cpb.ansrdgl = 150;
#if defined(E_W)
  /* Windows */
alloca_DOS_mem ( sizeof ( My_cpb ),
    & ( lp->cpb_DosPointer ),
    & ( lp->cpb.ProtectedSelector ),
    & ( lp->cpb.ProtectedPointer )
);
#endif

if ( lp->cpb.ProtectedSelector == 0 ) {
    Debug ( 3, sprintf ( Debug_buf,
        "LD_Dial/DoCallOutCommand:$t(PCS, $d bytes)",
        cDowMemErr, sizeof ( My_cpb ) )
    );
    ep->result = RC_RAMPFULL;
    lp->bSendEvent = TRUE;
    goto done;
}

memcpy ( ( void * ) lp->cpb.ProtectedPointer, /*dest*/ ( void * ) &My_cpb, /* source*/ sizeof ( My_cpb ) ) ;

#endif

nDialRetVal = setparm ( CHANNEL ( lp->nLineNum ),
#if defined(E_W)
    ( void * ) lp->cpb_DosPointer /* Windows */
#else
    &My_cpb
#endif
);

#if defined(E_W)
free_DOS_mem ( & ( lp->cpb_DosPointer ),
    & ( lp->cpb.ProtectedSelector ),
    & ( lp->cpb.ProtectedPointer ) );
#endif

switch ( nDialRetVal ) {
    case E_SUCCE: /* Function was successful. */
    /* success */
    break;
    case E_BADDL: /* D4x hardware error. */
    ep->result = RC_BADDLINE;
    lp->bSendEvent = TRUE;
    goto done;
}
default: /* some unknown error */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    goto done;
}

/* Ask channel to call out */

#if defined(E_W)  /* Windows */
 alloC DOS_mem ( 
    strlen ( cp->string ) + 1,
    &call DosPointer,  
    &call_ProtectedSelector,  
    &call_ProtectedPointer  
);

    if ( call_ProtectedSelector == 0 ) { 
        ep->result = RC_RAMFULL;
        lp->bSendEvent = TRUE;
        goto done;
    }

    memcpy ( 
        ( void * ) call_ProtectedPointer,  /* dest */
        ( void * ) cp->string,  /* source */
        strlen ( cp->string ) + 1
    );
#endif

nDialRetVal = callp ( 
    CHANNEL ( lp->nLineNum ),
    #if defined(E_W)
        ( void * ) call_DosPointer
    #else
        cp->string /* dial string */
    #endif
    
#ifdef(E_W)  /* Windows */
    free DOS_mem ( &call_DosPointer,
    &call_ProtectedSelector,
    &call_ProtectedPointer ) ;
#endif

switch ( nDialRetVal ) {
    case E_SUCCE: /* Function was successful. */
        /* success */
        break;
    case E_BADDL: /* D4x hardware error. */
        ep->result = RC_BADLINE;
        lp->bSendEvent = TRUE;
        goto done;
    default: /* some unknown error */
        ep->result = RC_INTERNALLINE;
        lp->bSendEvent = TRUE;
        goto done;
}

ip->nCmdStage = 1;
/* now wait until we get a call-answered event */
    goto done;

case 1:    /* we expect a call-answered event */
    IF NO EVENT THEN BREAK;
    IF WINDOWS EVENT THEN INTERNALFAIL;
nDialEvent = NEXT EVENT.nDialEvent;
nDialCallState = NEXT_EVENT.nDialCallState;
REMOVE EVENT;
switch ( nDialEvent ) {
    case T_SILOFF: /* Silence off. */
        lp->silent = FALSE;
        goto done; // remain in same state
    case T_SILOW: /* Silence on. */
        lp->silent = TRUE;
        goto done; // remain in same state
    case T_CATERM: /* Call analysis termination */
        /* restore usual settings */
        Set_Channel_Mask ( CEM_STANDARD, 1 );
        if ( lp->bSendEvent )
            goto done;
    switch ( nDialCallState ) {
        case CA_BUSY: /* Called line is busy. */
        case CA_OPINT: /* Called line recvd oper intercept */
            lp->nTempVRetVal = RC_BUSYDEST;
            break;
        case CA_NOAN: /* Called line did not answer. */
        case CA_NORNG: /* Called line did not ring. */
            lp->nTempVRetVal = RC_NOANSWER;
            break;
        case CA_CONN: /* Called line connected. */
            lp->bUserIsOffHook = TRUE;
            /* success */
            lp->bCardIsOffHook = TRUE;
            ep->result = RC_SUCCEEDED;
            if ( cp->wait_for_silence ) {
                ep->voice_after_answer = 0;
            }
        /* Get call analysis results */
        #if defined(E_W)
        /* Windows */
        alloc DOS_mem(
            sizeof ( My_car ),
            &call_DosPointer,
            &call_ProtectedSelector,
            &call_ProtectedPointer
        );
        #if defined(E_W)
        /* Windows */
        memcpy ( &My_car, /* source */
            ( void* ) call_ProtectedPointer,
            sizeof ( My_car )
        );
        free DOS_mem ( &call_DosPointer,

- 249 -

```c
&call_ProtectedSelector,
&call_ProtectedPointer);

#endif

if ( nDialRetVal != E_SUCC ) {
    ep->result = RC_INTERNALLINE;
} else {
    ep->voice_after_answer =
    My_car_ansrsize * 10;
}

lp->bSendEvent = TRUE;
goto done;
default: /* some unknown callstate */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
goto done;
}
break;
default: /* some unknown event */
    /* restore usual settings */
    Set_Channel_Mask ( CEM_STANDARD, 1 );
    if ( lp->bSendEvent )
        goto done;
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
goto done;
}

/* no answer and we are still off-hook; hang up */

/*
 * Ask channel to go on-hook.
 */
nDialRetVal = sethook ( CHANNEL ( lp->nLineNum ),
                        H_ONH
                        );

switch ( nDialRetVal ) {  
    case E_SUCC: /* Function was successful. */
        /* success */
        break;
    case E_BADDL: /* D4x hardware error. */
        ep->result = RC_BADLINE;
        lp->bSendEvent = TRUE;
goto done;
    default: /* some unknown error */
        ep->result = RC_INTERNALLINE;
        lp->bSendEvent = TRUE;
        goto done;
}

lp->nCmdStage = 2;
/* now wait until we get an on-hook-complete event */
goto done;

case 2:
    IF_NO_EVENT_THEN_BREAK;
    IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
nDialEvent = NEXT_EVENT.nDialEvent;
    REMOVE_EVENT;
    switch ( nDialEvent ) {
        case T_SILOFF: /* Silence off. */
        case T_SILON: /* Silence on. */
            goto done; // ignore; remain in same state
case T_ONH: /* Onhook complete. */
    break;
default: /* some unknown event */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    goto done;
}

// hung up successfully
// wait a few seconds to make sure CO sees we are hung up
lp->timeout = TIMEOUT_AFTER_ONHOOK;
REMOVE_TIMEOUTEVENTS;
lp->nCmdStage = 3;
/* now wait until we get a timeout event */
goto done;

case 3:
    IF_NO_EVENT THEN BREAK;
    if (TIMEOUT_EVENT)
        REMOVE_EVENT;
        // ignore it and keep waiting
        goto done;
    REMOVE_EVENT;
    /* make assumption */
    lp->bUserIsOffHook = FALSE;
    /* success */
    lp->bCardIsOffHook = FALSE;
    ep->result = lp->nTempVRetVal; // saved result code
    lp->bSendEvent = TRUE;
    goto done;

case SHUTDOWN_STAGE: // server is shutting down
    ep->result = RC_SERVERSHUTDOWN;
    lp->bSendEvent = TRUE;
    goto done;

default: // unknown command stage
    ep->result = RC_INTERNAL;
    lp->bSendEvent = TRUE;
    goto done;
}

done: // if complete, check to see if we have to do clean up
    if (lp->bSendEvent) {
        /* do something here */
        free_DOS_mem ( &call_DosPointer, &call_ProtectedSelector,
                      &call_ProtectedPointer );

        free_DOS_mem ( &(lp->cpb_DosPointer),
                       &(lp->cpb_ProtectedSelector),
                       &(lp->cpb_ProtectedPointer) );
    }
    return;

} // end function (DoCallOutCommand)

******************************************************************************
* DoConnectCommand -- Grab ownership of the current line.
*
static
void DoConnectCommand (void)
{
    CPB My_cpb;       // Channel parameters
    CSB My_csb;       // Channel status block
    int nDialRetVal;  // Dialogic return value

    switch (lp->nCmdStage) {
        case 0:  // start of command
            // fill in standard parts of event */
            REMOVE_ALL_EVENTS;
            ep->nEvent = EV_CONNECTED;
            ep->num_lines = nActualLines;    // valid even if cmd
            if ( nDialRetVal ) {
                ep->result = nDialRetVal;
                lp->bSendEvent = TRUE;
                goto done;
            }
            if ( ! ( My_csb.linestat & LS_HOOK ) ||    // line off
                 ! ( My_csb.linestat & LS_RING ) ||    // ring
                 ! ( My_csb.linestat & LS_SIL )        // silence
                 ! ( My_csb.linestat & LS_LOOP )        // loop current
            ) {
                ep->result = RC_BUSYLINE;
                lp->bSendEvent = TRUE;
                goto done;
            }
            // set channel parameters */
            cir cpb ( &My_cpb );
            #if defined(EBW) // Windows */
            alloc_DOS_mem (sizeof ( My_cpb ),
                           lp->cpb_Ptr,  // cpb
                           lp->cpb_Ptr,  // cpb_Ptr
                           lp->cpb_Ptr,  // cpb_Ptr
                           lp->cpb_Ptr,  // cpb_Ptr
            )
            if ( lp->cpb_Ptr == 0 ) {
                Debug (3, sprintf ( Debug_buf,
                                    "LD_Dial/DoConnectCommand:$s(CPB, %d bytes)",
                                    cDOSMemErr, sizeof ( My_cpb ) )
                );
                ep->result = RC_RAMFULL;
                lp->bSendEvent = TRUE;
                goto done;
            }
            memcpy (  // dest
            }
- 252 -

( void * ) & My_cpb, // source
sizeof ( My_cpb )
);
#endif

nDialRetVal = setparm ( // Windows
CHANNEL ( lp->nLineNum ),
#else
( void * ) lp->cpb_DosPointer
#endif

&My_cpb
);
#endif

free_DOS_mem ( & ( lp->cpb_DosPointer ),
& ( lp->cpb_ProtectedSelector ),
& ( lp->cpb_ProtectedPointer ) );
#endif

switch ( nDialRetVal ) {
 case E_SUCC: // Function was successful. */
  // success */
  Set_Channel Mask ( CEM_STANDARD, 1 );
  if ( lp->bSendEvent )
    goto done;
  lp->bUserIsOffHook = FALSE;
  lp->bCardIsOffHook = FALSE;
  lp->bConnected = TRUE;
  ep->result = RC_SUCCED;
  lp->bSendEvent = TRUE;
  goto done;
#endif

 case E_BADDL: // Bad hardware error. */
  ep->result = RC_BADLINE;
  lp->bSendEvent = TRUE;
  goto done;
  default: // some unknown error */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    goto done;
  }
  // never get here */

 case SHUTDOWN_STAGE: // server is shutting down
  ep->result = RC_SERVERSHUTDOWN;
  lp->bSendEvent = TRUE;
  goto done;
#endif

 default: // unknown command stage
  ep->result = RC_INTERNAL;
  lp->bSendEvent = TRUE;
  goto done;
  }

done: // if complete, check to see if we have to do clean up
  if (lp->bSendEvent) {
    // do something here !!!
  }

  return;

  // end function (DoConnectCommand)
**DoDisconnectCommand** -- Relinquish ownership of the current line.

```c
static
void DoDisconnectCommand (void)
{
    int ifile_num; /* indexed filenum (0 to MAX_IFILE-1) */
    int nDialRetVal; // Dialogic return value
    int nDialEvent; // Dialogic event number

    switch (lp->nCmdStage)
    {
      case 0: // start of command
        /* this is a multi-stage command */
        REMOVE_ALL_EVENTS;
        ep->nEvent = EV_DISCONNECTED;
        // close any indexed files that are open */
        for (ifile_num = 0; ifile_num < lp->num_openifile;
            ifile_num++)
        {
          if ((lp->ifile[ifile_num].ifhandle >= 0))
          {
            close (lp->ifile[ifile_num].ifhandle);
            lp->ifile[ifile_num].ifhandle = (-1);
          }
        }
        lp->num_openifile = 0;
        // hang up line
        nDialRetVal = sethook (CHANNEL (lp->nLineNum),
            H_ORH /* go on-hook */);
        }
        switch (nDialRetVal)
        {
          case E_SUC: /* Function was successful. */
            /* success */
            break;
          case E_BADDL: /* D4x hardware error. */
            ep->result = RC_BADLINE;
            lp->bSendEvent = TRUE;
            goto done;
          default: /* some unknown error */
            ep->result = RC_INTERNALFAIL;
            lp->bSendEvent = TRUE;
            goto done;
          }
        }
        lp->timeout = TIMEOUT_FOR_BROKEN;
        REMOVE_TIMEOUT_EVENTS;
        /* now wait until we get an on-hook-complete event */
        lp->nCmdStage = 1;
        goto done;

      case 1:
        IF_NO_EVENT_THEN_BREAK;
        IF_TIMEOUT_THEN_INTERNALFAIL;
        IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
        nDialEvent = NEXT_EVENT.nDialEvent;
        REMOVE_EVENT;
        switch (nDialEvent)
        {
          case T_SILOFF: /* Silence off. */
              case T_SILO: /* Silence on. */
```
- 254 -
        goto done;  // ignore; remain in same state
  case T_ONH:  /* Onhook complete. */
    /* success */
    break;
  default:     /* some unknown event */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    goto done;
  }
  Set_Channel_Mask ( 0, 0 );
  if ( lp->bSendEvent )
    goto done;
    // hung up successfully
  wait a few seconds to make sure CO sees we are hung up
  lp->timeout = TIMEOUT_AFTER_ONHOOK;
  REMOVE_TIMEOUT_EVENTS;
  lp->nCmdStage = 2;
    /* now wait until we get a timeout event */
    goto done;
  
case 2:
    IF_NO_EVENTTHEN_BREAK;
    if ( !TIMEOUT EVENT )
      REMOVE_EVENT;
        // ignore it and keep waiting
        goto done;
    }
    REMOVE_EVENT;
    /* make assumption */
    lp->bUserIsOffHook = FALSE;
    /* success */
    lp->bCardIsOffHook = FALSE;
    lp->bConnected = FALSE;
    ep->result = RC_SUCCEEDED;
    lp->bSendEvent = TRUE;
    goto done;

  case SHUTDOWN STAGE:  // server is shutting down
    ep->result = RC_SERVERSHUTDOWN;
    lp->bSendEvent = TRUE;
    goto done;
    
    default:  // unknown command stage
    ep->result = RC_INTERNAL;
    lp->bSendEvent = TRUE;
    goto done;
  }

done:
  return;
  }  // end function (DoDisconnectCommand)

/**************************************************************************
    * DoWaitForCallCommand -- Wait for an incoming call on  
    * the current line and answer it. 
    */
/**************************************************************************

static void DoWaitForCallCommand ( void)
{ 
  int nDialRetVal; // Dialogic return value
  int nDialEvent; // Dialogic event number
  CSB My_cscb;

  switch (lp->nCmdStage) {
    case 0: // start of command
      REMOVE_ALL_EVENTS;
      ep->nEvent = KV_INCOMINGCALL;
      if ( lp->bCardIsOffHook ) {
        ep->result = RC_CARDOFFHOOK;
        lp->bSendEvent = TRUE;
        goto done;
      }
      // Check line status
      nDialRetVal = Get_Line_Status( &My_cscb );
      if ( nDialRetVal ) {
        ep->result = nDialRetVal;
        lp->bSendEvent = TRUE;
        goto done;
      }
      if ( ! ( My_cscb.linestat & LS_HOOK ) ) || // line off
        // Not to detect no silence since at this moment, there may
        // be ring coming in.
        // ! ( My_cscb.linestat & LS_SIL ) || // silence
        ! ( My_cscb.linestat & LS_LOOP ) // loop current
        goto done;
      }
      ep->result = RC_BUSYLINE;
      lp->bSendEvent = TRUE;
      goto done;
    }
    // If ring already present, answer it
    if ( ! ( My_csbb.linestat & LS_RING ) ) {
      ADD_DIALOGIC_EVENT( T_RING, 0 )
    }
  }
  // Ask channel to report inbound calls.
  Set_Channel_Mask ( C_RING + CEM_STANDARD,
                      cp->answer_rings );
  if ( lp->bSendEvent )
    goto done;
  if ( cp->total_timeout == 0 )
    lp->timeout = 0; /* wait forever */
  else {
    lp->timeout = ( time_in_msec ( ) +
                  cp->total_timeout );
  }
  /* now wait until we get an incoming-call event */
  if ( lp->nCmdStage = 1;
       goto done;
  }
  case 1: // we expect an incoming-call event */
    IF_NO_EVENT_THEN_BREAK;
    if (TIMEOUT_EVENT) {
      REMOVE_EVENT;
      ep->result = RC_TOTALTIMEOUT;
      lp->bSendEvent = TRUE;
      goto done;
    }
  }
}
IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
nDialEvent = NEXT_EVENT.nDialEvent;
REMOVE_EVENT;

5
switch (nDialEvent)
{
    case T_SILOFF: /* Silence off. */
    case T_SILOM: /* Silence on. */
        goto done; // ignore; remain in same state
}

10
Set_Channel_Mask (CM_STANDARD, 1);
if (lp->bSendEvent)
goto done;

15
switch (nDialEvent)
{
    case T_RING: /* Rings received. */
        /* success */
        break;
    default: /* some unknown event */
        ep->result = RC_INTERNALLINE;
        lp->bSendEvent = TRUE;
        goto done;
}

20
/*
 * Ask channel to go off-hook.
 */

25
nDialRetVal = sethook (
    CHANNEL (lp->nLineNum),
    H_OFFH
);

switch (nDialRetVal)
{
    case E_SUCC: /* Function was successful. */
        /* success */
        break;
    case E_BADDL: /* D4x hardware error. */
        ep->result = RC_BADLINE;
        lp->bSendEvent = TRUE;
        goto done;
    default: /* some unknown error */
        ep->result = RC_INTERNALLINE;
        lp->bSendEvent = TRUE;
        goto done;
}

30
lp->timeout = TIMEOUT_FOR_BROKEN;
REMOVE_TIMEOUT_EVENTS;
/* now wait until we get an off-hook-complete event */

40
lp->nCmdStage = 2;
goto done;

45
case 2:
    IF_NO_EVENT_THEN_BREAK;
    IF_TIMEOUT_THEN_INTERNALFAIL;
    IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
nDialEvent = NEXT_EVENT.nDialEvent;
    REMOVE_EVENT;

switch (nDialEvent)
{
    case T_SILOFF: /* Silence off. */
    case T_SILOM: /* Silence on. */
        goto done; // ignore; remain in same state
    case T_OFFH: /* Offhook complete. */
        lp->bCardIsOffHook = TRUE;
        lp->bUserIsOffHook = TRUE;
        /* success */
        ep->result = RC_SUCCESSED;
- 257 -
lp->bSendEvent = TRUE;
goto done;
default: /* some unknown event */
ep->result = RC_INTERNALLINK;
lp->bSendEvent = TRUE;
goto done;
}
*/ never get here */
break;

case SHUTDOWN_STAGE: // server is shutting down
ep->result = RC_SERVERSHUTDOWN;
lp->bSendEvent = TRUE;
goto done;
default: // unknown command stage
ep->result = RC_INTERNAL;
lp->bSendEvent = TRUE;
goto done;
}
done:
return;
} // end function (DoWaitForCallCommand)

*******************************************************************************
* DoGetDTMFCommand -- Get DTMF digits from user.
*******************************************************************************
static void DoGetDTMFCommand (void)
{
RWB My_xrwb; // read/write block
struct findex far *fptr;
int nDialRetVal; // Dialogic return value
int nDialEvent; // Dialogic event number

switch (lp->nCmdStage) {
    case 0: // start of command
        ep->nEvent = EV_GETDTMF;
ep->string[0] = '\0';
if ( !lp->bCardIsOffHook ) {
    ep->result = RC_CARDONHOOK;
lp->bSendEvent = TRUE;
goto done;
} if ( !lp->bUserIsOffHook ) {
    ep->result = RC_USERONHOOK;
lp->bSendEvent = TRUE;
goto done;
} /*
    nDialRetVal = Check_User_Connection();
if ( nDialRetVal ) {
    ep->result = nDialRetVal;
lp->bSendEvent = TRUE;
}
goto done;
}
*/

if ( cp->flush_digits_at_start ) {
    Flush_All_DTMF ( lp->bSendEvent);
    if ( lp->bSendEvent )
        goto done;
}

Set_End_Conditions ( );
if ( lp->bSendEvent ) goto done;

// get digits
clrxxwb ( &My_xrwb );
/* set parameters */
if ( cp->end_oper_on_any_digit )
    My_xrwb.termdtmf = '0';
else if ( cp->end_oper_on_digit_end )
    My_xrwb.termdtmf = cp->end_digit;
else
    My_xrwb.termdtmf = '\0';
/* tricky: 0==unlimited(MAX_DIGITS); also handle too big */
My_xrwb.maxdtmf = lp->nMaxDTMF;
My_xrwb.issrwb = 1;
My_xrwb.maxsec = ( word ) MSEC_TO_SEC ( cp->total_timeout );

/*
 * maxsil == add up all silence between digits -- not
 * what we want
 */
/* My_xrwb.maxsil = MSEC_TO_SEC(cp->start_timeout); */
My_xrwb.intrdig = ( byte ) MSEC_TO_SEC ( cp->interdigit_to );
My_xrwb.rwbflag = 0;
My_xrwb.loopsig = 1;/* terminate on hangup */
#if defined(E_W)
/* Windows */
alloc_DOS_mem( MAX_DIGITS + 1,
               &lp->dtmf_DosPointer,
               &lp->dtmf_ProtectedSelector,
               &lp->dtmf_ProtecedSelector )
#endif
if ( lp->dtmf_ProtectedSelector == 0 ) {
    ep->result = RC_RAMFULL;
    lp->bSendEvent = TRUE;
    goto done;
}
#else
fpotr = ( void * ) lp->dtmf_DosPointer;
#endif
#if defined(E_W)
/* Windows */
alloc_DOS_mem ( sizeof ( My_xrwb ),
                &lp->xrwb_DosPointer ),
- 259 -

&( lp->xrbw_ProtectedSelector ),
&( lp->xrbw_ProtectedPointer )
);

if ( lp->xrbw_ProtectedSelector == 0 ) {
  Debug ( 3, sprintf ( Debug_buf,
     "ID_Dial/DoGetDTMFCommand:%s(XRWB, %d bytes)",
     cDtmfMemErr , sizeof ( My_xrbw ) ) );
  ep->result = RC_RAMPFULL;
  lp->bSendEvent = TRUE;
  goto done;
}

memcpy ( ( void * ) lp->xrbw_ProtectedPointer,
( void * ) &My_xrbw , sizeof ( My_xrbw ) );
#endif

/* start recording data */

nDialRetVal = getdtmfs (  
  CHANNEL ( lp->nLineNum ),  
  /* Windows */  
  &RWB ,  
  /* non-Windows */  
  &My_xrbw
  
  switch ( nDialRetVal ) {  
  case E_SUCCEED: /* Function was successful. */
    break;
  default:  /* some unknown error */
    ep->result = RC_INTERNALFAIL;
    lp->bSendEvent = TRUE;
    goto done;
  }
  /* now wait until we get an event */
  lp->nCmdStage = 2;
  goto done;
}

case 2: /* we expect an end-of-input event */

  IF_NO_EVENT_THEN_BREAK;
  IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
  nDialEvent = NEXT_EVENT, nDialEvent;
  REMOVE_EVENT;
  switch ( nDialEvent ) {
  case T_SILOFF: /* Silence off. */
  case T_SILON: /* Silence on. */
    goto done;  // ignore; remain in same state
  }
#endif

memcpy ( dial_dtmfbuf[lp - line],
( void * ) lp->dtmf_ProtectedPointer,
  MAX_DIGITS + 1 );

free_DOS_mem ( &( lp->xrbw_DosPointer ),
  &( lp->xrbw_ProtectedSelector ),
  &( lp->xrbw_ProtectedPointer ) );

  free_DOS_mem ( &( lp->dtmf_DosPointer ),
  &( lp->dtmf_ProtectedSelector ),
  &( lp->dtmf_ProtectedPointer ) );

switch ( nDialEvent ) {
  case T_MAXDIG: /* Maximum DTMF digits received. */
    ep->result = RC_DATALENGTH;
    strcpy ( ep->string, dial_dtmfbuf[lp - line] );
- 260 -

lp->bSendEvent = TRUE;
goto done;
case T_TERMDT:    /* Terminating DTMF digit
      received. */

5    case T_MDTMF:    /* Terminated by masked DTMF digit*/
    ep->result = RC_ENDFLAG;
    strcpy ( ep->string, dial_dtmbuf[lp - line] );
    lp->bSendEvent = TRUE;
goto done;

10    case T_TIME:    /* Rec/play/getdtmf timed out. */
    case T_SIL:     /* Maximum silence received. */
    case T_IDTIME:  /* Interdigit delay exceeded. */
    strcpy ( ep->string, dial_dtmbuf[lp - line] );
    if ( ep->string[0] == '\0' )
      ep->result = RC_STARTTIMEOUT;
    else
      ep->result = RC_INTERDIGITTIMEOUT;
    lp->bSendEvent = TRUE;
goto done;

20    case T_LCTERM:  /* Terminate by drop in loop
      signal */
      ep->result = RC_USERHUNGUP;
      strcpy ( ep->string, dial_dtmbuf[lp - line] );
    lp->bSendEvent = TRUE;
goto done;

25    case T_STOP:    /* Rec/play/getdtmf stopped. */
    case T_HFAIL:   /* Hardware failure. */
    ep->result = RC_BADLINE;
    lp->bSendEvent = TRUE;
goto done;

30    default:       /* some unknown event */
      ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
goto done;

35    }/* never get here */
break;

35    case SHUTDOWN_STAGE: // server is shutting down
40      ep->result = RC_SERVERSHUTDOWN;
    lp->bSendEvent = TRUE;
goto done;

45    default:        // unknown command stage
      ep->result = RC_INTERNAL;
    lp->bSendEvent = TRUE;
goto done;
    }

50    done:
    // if complete, check to see if we have to do clean up
    if (lp->bSendEvent) {
      // do something here !!!
      free_DOS_mem ( & (lp->xrbx_DosPointer ),

55      & (lp->xrbx_ProtectedSelector ),
      & (lp->xrbx_ProtectedPointer ) );
      free_DOS_mem ( & (lp->dtmf_DosPointer ),
      & (lp->dtmf_ProtectedSelector ),
      & (lp->dtmf_ProtectedPointer ) );

60    } return;    // end function (DoGetDTMFCommand)
/***********************
* DoGetWordsCommand -- Get words from user.
*************************/

static void DoGetWordsCommand (void)
{
    int nDialEvent;     // Dialogic event number
    int nDialCallState; // Dialogic call state number

    switch (lp->nCmdStage) {
        case 0:      // start of command
            ep->nEvent = EV_GETWORDS;
            ep->string[0] = '\0';
            /* error-checking */
            if ( !lp->bCardIsOffHook ) {
                ep->result = RC_CARDONHOOK;
                lp->bSendEvent = TRUE;
                goto done;
            }
            if ( !lp->bUserIsOffHook ) {
                ep->result = RC_USERTONHOOK;
                lp->bSendEvent = TRUE;
                goto done;
            }
            lp->nCmdStage = 40;
            // tricky; fall through ...
        case 40:     // still start of command
            vr_Get_Words ( *cp,
                            0, // nDialEvent
                            0, // nDialCallState
                            &lp->nCmdStage,
                            ep,
                            &lp->bSendEvent
                        );
            goto done;
        case 41:     // some event
        case 42:     // some event
            IF_NO_EVENT THEN BREAK;
            IF_WINDOWS_EVENT THEN INTERNALFAIL;
            nDialEvent = NEXT_EVENT.nDialEvent;
            nDialCallState = NEXT_EVENT.nDialCallState;
            REMOVE_EVENT;
            vr_Get_Words ( *cp,
                            NEXT_EVENT.nDialEvent,
                            NEXT_EVENT.nDialCallState,
                            &lp->nCmdStage,
                            ep,
                            &lp->bSendEvent
                        );
            goto done;
        case SHUTDOWN_STAGE: // server is shutting down

vr_Get_Words {
    *cp,
    0,
    0,
    &lp->nCmdStage,
    ep,
    &lp->bSendEvent
};
goto done;

default:    // unknown command stage
    ep->result = RC_INTERNAL;
    lp->bSendEvent = TRUE;
    goto done;
}

done:
    return;
}    // end function (DoGetWordsCommand)

/************************************************************
* DoSetIFileCommand    --  Open indexed prompt file.
* ....................................................................

static
void DoSetIFileCommand (void)
{
    int ifile_num;    // indexed filenum (0 to MAX_IFILE-1) */
    int pcount;      // phrase count */
    char buffer[40]; // file input buffer for setifile */
    int rval;
    char *ptr, *startptr, *endptr;

    // in-file struct to access an indexed file */
    struct {
        long max;    // number of if_entry's that follow
        if_header */
        long rate;    // always 6053 */
        long top;     // number of phrases in file (<= max) */
        long rsv3;   // reserved */
        long size;    // size of file in bytes */
        long rsv5;   // reserved */
    } if_header;

    // in-file struct to access an indexed file */
    // one if_entry per phrase in file */
    struct {
        long start;    // starting byte in file */
        long length;   // length in bytes (0==no phrase here)*/
        long annotate; // unused (== 0) */
    } if_entry;

    switch (lp->nCmdStage) {
        case 0:    // start of command
            ep->nEvent = EV_SETIFILE;
            /* error-checking */
            if ( lp->num_openifile >= MAX_IFILE ) {
ep->result = RC_BADCOMMAND;
lp->bSendEvent = TRUE;
goto done;
}

ifile_num = lp->num_openifile;

/*
 * First, do the voice file.
 */

lp->ifile[ifile_num].ifhandle = open ( 
cp->play_item.fname, O_BINARY | O_RDONLY );
if ( lp->ifile[ifile_num].ifhandle < 0 ) {
  ep->result = RC_NOPFILE;
  ep->C_error = errno;
  ep->DOS_error = doserrno;
  lp->bSendEvent = TRUE;
  goto done;
}

/* read header from indexed file */
rval = read ( 
  lp->ifile[ifile_num].ifhandle, 
  if_header, 
  sizeof ( if_header )
);

if ( rval < sizeof ( if_header ) ) {
  ep->result = RC_BADFILE;
  ep->C_error = errno;
  ep->DOS_error = doserrno;
  close ( lp->ifile[ifile_num].ifhandle );
  lp->ifile[ifile_num].ifhandle = (-1);
  lp->bSendEvent = TRUE;
  goto done;
}

/* do some error-checking */
if ( ( if_header.top < 0 ) || ( if_header.max < 0 ) ) {
  ep->result = RC_BADFILE;
  close ( lp->ifile[ifile_num].ifhandle );
  lp->ifile[ifile_num].ifhandle = (-1);
  lp->bSendEvent = TRUE;
  goto done;
}

if ( if_header.max >= MAX_PHRASE ) {
  ep->result = RC_BADFILE;
  close ( lp->ifile[ifile_num].ifhandle );
  lp->ifile[ifile_num].ifhandle = (-1);
  goto done;
}

/* read phrase entries from indexed file */
for ( pcount = 0; pcount < if_header.max; pcount++ ) {
  rval = read ( 
    lp->ifile[ifile_num].ifhandle, 
    if_entry, 
    sizeof ( if_entry )
  );
  if ( rval < sizeof ( if_entry ) ) {
    ep->result = RC_BADFILE;
    ep->C_error = errno;
    ep->DOS_error = doserrno;
    close ( lp->ifile[ifile_num].ifhandle );
  }
- 264 -

lp->ifile[ifile_num].ifhandle = ( -1 );
lp->bSendEvent = TRUE;
goto done;
}

/* do some error-checking */
if ( ( if_entry.start < ( sizeof ( if_header ) + ( if_header.max + sizeof ( if_entry ) ) ) )
    || ( if_entry.length < 0 ) ) {
    ep->result = RC_BADFILE;
    close ( lp->ifile[ifile_num].ifhandle );
    lp->ifile[ifile_num].ifhandle = ( -1 );
    lp->bSendEvent = TRUE;
    goto done;
}

lp->ifile[ifile_num].indextab[pcount].start =
    if_entry.start;
lp->ifile[ifile_num].indextab[pcount].length =
    if_entry.length;

lp->ifile[ifile_num].num_phrases = ( int )
    if_header.max;

/*
 * Second, do the map file.
 */
lp->open_file = open ( cp->name,
    O_RDONLY | O_RDONLY
);

if ( lp->open_file < 0 ) {
    ep->result = RC_NOPFILE;
    ep->C_error = errno;
    ep->DOS_error = _doserrno;
    close ( lp->ifile[ifile_num].ifhandle );
    lp->ifile[ifile_num].ifhandle = ( -1 );
    lp->bSendEvent = TRUE;
    goto done;
}

/* read phrase file names (1 per line) from map file */
pcount = 0;
lp->ifile[ifile_num].indextab[pcount].fname[0] = '\0';
while ( pcount < lp->ifile[ifile_num].num_phrases ) {
    rval = read ( lp->open_file,
        &buffer,
        sizeof ( buffer ) - 1
    );
    if ( rval == 0 ) {
        /* EOF before proper number of phrases done */
        close ( lp->ifile[ifile_num].ifhandle );
        lp->ifile[ifile_num].ifhandle = ( -1 );
        lp->open_file = ( -1 );
        ep->result = RC_BADFILE;
        lp->bSendEvent = TRUE;
        goto done;
    }
    if ( rval < 0 ) {
        close ( lp->ifile[ifile_num].ifhandle );
        lp->ifile[ifile_num].ifhandle = ( -1 );
        close ( lp->open_file );
        lp->open_file = ( -1 );
    }
}

ep->result = RC_BADFILE;
ep->C_error = errno;
ep->DOS_error = _doserrno;
lp->bSendEvent = TRUE;
goto done;
}

buffer[rval] = '\0'; /* for debugging purposes */
ptr = buffer;
startptr = ptr;
endptr = ptr + rval;
while ( ( ptr < endptr ) && ( *ptr != '\n' ) )
    ptr++;
strncat ( lp->ifile[ifile_num].indextab[pcount].fname,
        startptr,
        ptr - startptr );
    if ( *ptr == '\n' ) {
        pcount++;
        ptr++;
        startptr = ptr;
    }
}
/* doesn't detect map file contains more lines than expected */
close ( lp->open_file );
lp->open_file = (-1);
if ( Debug_flag >= 9 )
    for ( pcount = 0; pcount < lp->ifile[ifile_num].num_phrases; pcount++ )
        lp->num_openifile++;
ep->result = RC_SUCCEEDED;
lp->bSendEvent = TRUE;
goto done;

default: // unknown command stage
    ep->result = RC_INTERNAL;
lp->bSendEvent = TRUE;
goto done;
}
done:
    return;
} // end function (DoSetIFileCommand)

/****************************
 * DoRecordCommand -- Record from the current line to a file.
 ****************************/

static
void DoRecordCommand (void)
{
    RWB    My_xrwb;       // read/write block
    int    nDialRetVal;   // Dialogic return value
    int    nDialEvent;    // Dialogic event number
    char   far    *fptr;

switch (lp->nCmdStage) {
    case 0: // start of command
        ep->nEvent = EV_RECORD;
        ep->string[0] = '\0';
        if ( !lp->bCardIsOffHook ) {
            ep->result = RC_CARDONHOOK;
            lp->bSendEvent = TRUE;
            goto done;
        }
        if ( !lp->bUserIsOffHook ) {
            ep->result = RC_USERONHOOK;
            lp->bSendEvent = TRUE;
            goto done;
        }
        /*
         * nDialRetVal = Check_User_Connection();
         * if ( nDialRetVal ) {
         *     ep->result = nDialRetVal;
         *     lp->bSendEvent = TRUE;
         *     goto done;
         * }
         */
        if ( cp->flush_digits_at_start ) {
            Flush_All_DTMF ( );
            if ( lp->bSendEvent )
                goto done;
        }
    }
    Set_End_Conditions ( );
    if ( lp->bSendEvent ) goto done;
    clrxxwb ( &My_xrwb );
    /* create the file and open it */
    lp->open_file = open ( cp->play_item.fname,
        O_CREAT | O_TRUNC | O_BINARY | O_RDWR, S_IRUSR | S_IWUSR );
    if ( lp->open_file < 0 ) {
        ep->result = RC_NOPFILE;
        ep->c_errno = errno;
        ep->DOS_errno = doserrno;
        lp->bSendEvent = TRUE;
        goto done;
    }
    My_xrwb.filehnd = lp->open_file;
    /* set parameters */
    /* total duration (secs) */
    My_xrwb.maxsec = ( word ) MSEC_TO_SEC ( cp->total_recording );
    /* max silence duration (secs) */
    My_xrwb.maxsil = ( byte ) MSEC_TO_SEC ( ( int ) ( cp->end_silence ) );
    if ( cp->end_voice_on_any_digit )
        My_xrwb.termdtmf = '\0';
    else if ( cp->end_voice_on_digit_end )
        My_xrwb.termdtmf = cp->end_digit;
    else
        My_xrwb.termdtmf = '\0';
    // My_xrwb.maxdtmf = lp->nMaxDTMF;
My_xrwb.rwbflags = RW_TONE; /* play beep tone */
My_xrwb.rwbdata1 = 2; /* beep duration (200 msec
    incs) */
My_xrwb.loopsig = 1; /* terminate on hangup */
My_xrwb.isrxb = 1;
My_xrwb.intrdig = ( byte ) MSEC_TO_SEC ( ( int ) ( 
    cp->end_silence ) );
My_xrwb.maxnsil = ( byte ) MSEC_TO_SEC ( ( int ) ( 
    cp->total_recording ) );

/* start recording data */
#if defined(E_W)
    /* Windows */
    alloc_DOS_mem ( 
        sizeof ( My_xrwb ),
        & ( lp->xrwb_DosPointer ),
        & ( lp->xrwb_ProtectedSelector ),
        & ( lp->xrwb_ProtectedPointer ) );
    if ( lp->xrwb_ProtectedSelector == 0 ) {
        ep->result = RC_RAMPFULL;
        lp->bSendEvent = TRUE;
        goto done;
    }
    memcpy ( ( void * ) lp->xrwb_ProtectedPointer,
        ( void * ) &My_xrwb, sizeof ( My_xrwb ) );
#endif

nDialRetVal = recfile ( CHANNEL ( lp->nLineNum ),
    #if defined(E_W)
        ( RWB * ) lp->xrwb_DosPointer, /* Windows */
    #else
        &My_xrwb, /* non-Windows */
    #endif
        RM_NORM
    );

switch ( nDialRetVal ) {
    case E_SUCCE: /* Function was successful. */
        break;
    case E_BADDL: /* D4x hardware error. */
        close ( lp->open_file );
        lp->open_file = ( -1 );
        ep->result = RC_BADLINE;
        lp->bSendEvent = TRUE;
        goto done;
    default: /* some unknown error */
        close ( lp->open_file );
        lp->open_file = ( -1 );
        ep->result = RC_INTERNALLINE;
        lp->bSendEvent = TRUE;
        goto done;
}

/* now wait until we get an event */
lp->nCmdStage = 1;
goto done;

case 1: /* we expect an end-of-recording event */
    IF_NO_EVENT_THEN_BREAK;
    IF WINDOWS_EVENT THEN_INTERNALFAIL;
    nDialEvent = NEXT_EVENT.nDialEvent;
    REMOVE_EVENT;
switch ( nDialEvent ) {
    case T_SILOFF: /* Silence off. */
    case T_SILON: /* Silence on. */
goto done; // ignore; remain in same state
}
#if defined(E_W)
  free_DOS_mem ( ( lp->xrwb_DosPointer ),
                ( lp->xrwb_ProtectedSelector ),
                ( lp->xrwb_ProtectedPointer ) );
#endif
close ( lp->open_file );
lp->open_file = (-1);
/* extract base filename from full filename */
fname_full_to_base ( cp->play_item.fname, /* input -- full filename */
                     ep->filename /* retval -- base fname */);
switch ( nDialEvent ) {
  case T_MAXDT: /* Maximum DTMF digits received. */
    lp->nTempVRetVal = RC_DATALength;
    break;
  case T_TERMDT: /* Terminating DTMF digit received. */
  case T_MDTMF: /* Terminated by masked DTMF digit */
    lp->nTempVRetVal = RC_TERMFLAG;
    break;
  case T_MAXBYT: /* Max bytes reached on play or rec */
    lp->nTempVRetVal = RC_DATALength;
    break;
  case T_TIMIE: /* Rec/play/getdtmf timed out. */
    lp->nTempVRetVal = RC_TIMEOUT;
    break;
  case T_SIL: /* Maximum silence received. */
  case T_IDTIME: /* Interdigit delay exceeded. */
    lp->nTempVRetVal = RC_INTERDIGITTIMEOUT;
    break;
  case T_LCTERM: /* Terminate by drop in loop signal */
    ep->result = RC_USERHUNGUP;
    // don’t bother getting any digits they may have entered (?)
    lp->bSendEvent = TRUE;
    goto done;
  case T_DOSERR: /* Dos error */
  case T_DFULL: /* Disk full */
    remove ( ep->filename );
    ep->filename[0] = ‘\0’;
    ep->result = RC_DISKFULL;
    lp->bSendEvent = TRUE;
    goto done;
  case T_STOP: /* Rec/play/getdtmf stopped. */
  case T_HFAIL: /* Hardware failure */
  case T_EMSERR: /* Terminated by EMS error */
    ep->filename[0] = ‘\0’;
    ep->result = RC_BADLINE;
    lp->bSendEvent = TRUE;
    goto done;
#if 0
  case 37: /* some undocumented event */
    /* ignore event */
    goto done;
#endif
  default: /* some unknown event */
    ep->filename[0] = ‘\0’;
ep->result = RC_INTERNALLINE;
lp->bSendEvent = TRUE;
goto done;
}

if (!cp->get_digits_at_end) {
    ep->result = lp->nTempVRSetVal;
lp->bSendEvent = TRUE;
goto done;
}

// get digits
clrxrw ( &My_xrw );
/* set parameters */
if ( cp->end_oper_on_any_digit )
    My_xrw.termdtmf = '0';
else if ( cp->end_oper_on_digit_end )
    My_xrw.termdtmf = cp->end_digit;
else
    My_xrw.termdtmf = '\0';
/* tricky: O=unlimited(MAX_DIGITS); also handle too big */
    My_xrw.maxdtmf = lp->nMaxDTMF;
    My_xrw.isxrw = 1;
    cp->total_timeout = SEC_TO_MSEC ( 1 ); /* don't wait*/
    My_xrw.maxsec = ( word ) MSEC_TO_SEC ( cp->total_timeout );

    /*
    maxsil = add up all silence between digits -- not what we
    * want
    */
    /* My_xrw.maxsil = MSEC_TO_SEC(cp->start_timeout); */
    My_xrw.intrdig = ( byte ) MSEC_TO_SEC ( cp->interdigit_to );
    My_xrw.rwbflags = 0;

    My_xrw.loopsig = 1; /* terminate on hangup */
#if defined(F_W)
    /* Windows */
    alloc_DOS_mem ( MAX_DIGITS + 1,
        ( lp->dtmf_DosPointer ),
        ( lp->dtmf_ProtectedSelector ),
        ( lp->dtmf_ProtectedPointer )
    );
    if ( lp->dtmf_ProtectedSelector == 0 ) {
        ep->result = RC_RAMFULL;
lp->bSendEvent = TRUE;
goto done;
    }
#else
    fptr = ( void *) lp->dtmf_DosPointer;
#endif
    fptr = dial_dtmobuf(lp->nLineNum);
#if defined(F_W)
    alloc_DOS_mem ( sizeof ( My_xrw ),
        ( lp->xrw_DosPointer ),
        ( lp->xrw_ProtectedSelector ),
        ( lp->xrw_ProtectedSelector )
    );
#endif
& ( lp->xrwbProtectedPointer )
}
if ( lp->xrwbProtectedSelector == 0 ) {
    ep->result = RC_RAMFULL;
    lp->bSendEvent = TRUE;
    goto done;
}
memcpy (( void *) lp->xrwbProtectedPointer,
        ( void *) &My_xrwb, sizeof ( My_xrwb ) );
#endif

/*@ start recording data */
#define DialRetVal = getdtmfs ( CHANNEL ( lp->nLineNum ),
#if defined(E_W)
        ( RWB *) lp->xrwb_DosPointer /* Windows */
#else
        &My_xrwb /* non-Windows */
#endif
switch ( DialRetVal ) {
    case E_SUCCE: /* Function was successful. */
    break;
    default: /* some unknown error */
    ep->result = RC_INTERNALLINE;
    lp->bSendEvent = TRUE;
    goto done;
} /* now wait until we get an event */
lp->nCmdStage = 2;
goto done;

    case 2: /* we expect an end-of-input event */
    IF_NO_EVENT_THEN_BREAK;
    IF_WINDOWS_EVENT_THEN_INTERNALFAIL;
    nDialEvent = NEXT_EVENT. nDialEvent;
    REMOVE_EVENT;
    switch ( nDialEvent ) {
    case T_SILOFF: /* Silence off. */
    case T_SILON: /* Silence on. */
    goto done; // ignore; remain in same state
    }
#endif
    memcpy ( dial_dtmfbuf[lp - line], ( void * )
        lp->dtmfProtectedPtr, MAX_DIGITS + 1 );
    free_DOS_mem ( & ( lp->xrwb_DosPointer ),
        & ( lp->xrwbProtectedSelector ),
        & ( lp->dtmf_DosPointer ),
        & ( lp->dtmfProtectedSelector ));
#endif
    switch ( nDialEvent ) {
    case T_MAXDDT: /* Maximum DTMF digits received. */
    ep->result = RC_DATALENGTH;
    strcpy ( ep->string, dial_dtmfbuf[lp - line] );
    lp->bSendEvent = TRUE;
    goto done;
    case T_TERMDDT: /* Terminating DTMF digit received. */
    case T_MDTMF: /* Terminated by masked DTMF digit */
    ep->result = RC_ENDFLAG;
    strcpy ( ep->string, dial_dtmfbuf[lp - line] );
lp->bSendEvent = TRUE;
goto done;
case T_TIME: /* Rec/play/getdtmf timed out. */
case T_SIL: /* Maximum silence received. */
case T_IDTIME: /* Interdigit delay exceeded. */
    strcpy ( ep->string, dial_dtmobuf[lp - line] );
    if ( ep->string[0] == '\0' )
        ep->result = RC_STARTTIMEOUT;
else
    ep->result = RC_INTERDIGITTIMEOUT;
lp->bSendEvent = TRUE;
goto done;
case T_LCTERM: /* Terminate by drop in loop signal */
    strcpy ( ep->string, dial_dtmobuf[lp - line] );
    lp->bSendEvent = TRUE;
goto done;
case T_STOP: /* Rec/play/getdtmf stopped. */
case T_HFAIL: /* Hardware failure. */
    ep->result = RC_BADLINE;
    lp->bSendEvent = TRUE;
goto done;
default: /* some unknown event */
    ep->result = RC_INTERNALLINE;
lp->bSendEvent = TRUE;
goto done;
}
/* never get here */
break;

case SHUTDOWN_STAGE: // server is shutting down
    ep->result = RC_SERVERSHUTDOWN;
    lp->bSendEvent = TRUE;
    goto done;
default: // unknown command stage
    ep->result = RC_INTERNAL;
    lp->bSendEvent = TRUE;
    goto done;
}
done:
/* if complete, check to see if we have to do clean up
45  if (lp->bSendEvent) {
46      // do something here !!!!
    free_DOS_mem ( &( lp->xrbw_DosPointer ),
        &( lp->xrbw.ProtectedSelector ),
        &( lp->xrbw.ProtectedPointer ) );
    free_DOS_mem ( &( lp->dtmf_DosPointer ),
        &( lp->dtmf.ProtectedSelector ),
        &( lp->dtmf.ProtectedPointer ) );
47  }
48  return;
49  } // end function (DoRecordCommand)
60
/*****************************/
#if DO_FAKE_LINES
// tricky: include C code into this file, because needs to get at
// line array
#include "fakeline.c"
#endif

/************************************************************************
*  ld_Do_Command
**************************************************************************/

int  ld_Do_Command (const struct command  cmd,  // command to execute
               const boolean  start_of_cmd,  // start new cmd ?
               or continue old
               struct event  *event  // result of command )
{
    int nVRetVal;  // Voysys return value
    boolean GotEvent;

    if (!Initialized) {
        nVRetVal = RC_NOTINITIALIZED;
        event->result = nVRetVal;
        goto done;
    }

    if ( ( cmd.nLine < 0 )

    #if DO_FAKE_LINES
    || ( cmd.nLine >= ( int ) MAX_AL ) ) {
    #else
    || ( cmd.nLine >= ( int ) nActualLines ) ) {
    #endif
        nVRetVal = RC_NOLINE;
        event->result = nVRetVal;
        goto done;
    }

    // see if driver has any events for ANY line - get all of them */
    GotEvent = TRUE;
    while ( GotEvent )
        Check_For_Event ( &GotEvent );

    lp = &line[cmd.nLine];
    cp = &lp->cmd ;
    ep = &lp->event ;
    ep->result = RC_CMDNOTDONE;

    // we are waiting for an event and we haven't got one yet, // return */
    if ( ( !start_of_cmd ) && ( !IS_AN_EVENT ) ) {
        if ( ( ep->bCmdInProgress ) && ( ep->nCmdStage ==
            ABORT_STAGE))
            goto doit;
        if ( ( lp->bCmdInProgress ) && ( lp->nCmdStage ==
            SHUTDOWN_STAGE))
            goto doit;
        if ( ( lp->timeout != 0 ) && ( time_in_msec() >=
            lp->timeout ) ) { // command timed out */
            ADD_EVENT(WM_TIMER,0,0,0);
5
if (start_of_cmd)
lp->nCmdStage = 0;
doit:
10
lp->bSendEvent = FALSE;

// handle case of new command */
if (start_of_cmd) {
    ep->nEvent = EV_NONE;
    ep->nLine = cmd.nLine;
    ep->C_error = 0;
    ep->DV_error = 0;
    ep->DOS_error = 0;
    ep->string[0] = '\0';
    ep->filename[0] = '\0';
    if (lp->bCmdInProgress && (cmd.nCommand == DVRC_ABORT)) {
        // handle abort by putting current command into funny stage */
        lp->nCmdStage = ABORT_STAGE;
    } else {
        if (lp->bCmdInProgress) {
            // error -- new command before previous command finished */
            ep->nEvent = EV_BUSYWITHCMD;
            ep->result = RC_BUSYLINE;  // not quite right
            goto sendev;
        }
    }
    lp->cmd = cmd;
    lp->bCmdInProgress = TRUE;
    lp->nCmdStage = 0;
    lp->timeout = 0;
}

#if DO_FAKE_LINES
if (cmd.nLine >= (int)nActualLines) {
        DoFakeLineCommand();
        if (lp->bSendEvent)
            goto sendev;
        goto finished;
}
#endif

switch (cp->nCommand) {

    /**************************************************************************
    case DVRC_CONNECT:       // connect Line task to line N
    DoConnectCommand();
    if (lp->bSendEvent)
        goto sendev;
    goto finished;

    /**************************************************************************
    case DVRC_DISCONNECT:    // disconnect Line task from line N
    DoDisconnectCommand();
    if (lp->bSendEvent)
        goto sendev;
    goto finished;
*/
case DVRC_GETSTATE:
    ep->nEvent = EV_GOTSTATE;
    ep->user_off_hook = lp->bUserIsOffHook;
    ep->result = RC_SUCCEEDED;
    goto sendev;

case DVRC_WAITFORCALL: /* wait for incoming call +
     answer it */
    DoWaitForCallCommand();
    if (lp->bSendEvent)
        goto sendev;
    goto finished;

case DVRC_RECORD: /* record voice or FAX from user */
    DoRecordCommand();
    if (lp->bSendEvent)
        goto sendev;
    goto finished;

case DVRC_INITPLAY: /* initialize playlist to empty */
    ep->nEvent = EV_INITPLAY;
    lp->plist_count = 0;
    ep->result = RC_SUCCEEDED;
    goto sendev;

case DVRC_ADDPLAY: /* add item to playlist */
    ep->nEvent = EV_ADDPLAY;
    if (lp->plist_count >= MAX_PF) {
        ep->result = RC_PLISTLENGTH;
        goto sendev;
    }
    if (cp->play_item.type != PI_SINGLE_FILE) {
        ep->result = RC_UNIMPCOMMAND;
        goto sendev;
    }
    lp->plist[lp->plist_count] = cp->play_item;
    /* see if we can find prompt in an indexed file */
    Find_Phase_In_File ( &lp->plist[lp->plist_count]);
    lp->plist_count++;
    ep->result = RC_SUCCEEDED;
    goto sendev;

case DVRC_PLAY: /* play voice or FAX out to user */
    DoPlayCommand();
    if (lp->bSendEvent)
        goto sendev;
    goto finished;

case DVRC_CALLOUT: /* initiate a call + wait for answer */
    DoCallOutCommand();
    if (lp->bSendEvent)
        goto sendev;
    goto finished;
case DVRC_GETDTMF: /* get DTMF digit string from user */
    DoGetDTMFCommand();
    if ( lp->bSendEvent )
        goto sendev;
    goto finished;

case DVRC_SENDDTMPF: /* send DTMF digit string out on line */
    DoSendDTMFCommand();
    if ( lp->bSendEvent )
        goto sendev;
    goto finished;

case DVRC_ABORT: /* abort any operation in progress */
    /* if we get here, it means there was no command in progress */
    ep->nEvent = EV_ABORT;
    ep->result = RC_NULLABORT;
    goto sendev;

case DVRC_HANGUP: /* hang up (go on-hook) */
    DoHangupCommand();
    if ( lp->bSendEvent )
        goto sendev;
    goto finished;

case DVRC_SETIFILE: /* open indexed prompt file */
    DoSetIFileCommand();
    if ( lp->bSendEvent )
        goto sendev;
    goto finished;

case DVRC_SETVOCAB: /* open vocabulary map files */
    ep->nEvent = EV_SETVOCAB;
    ep->result = vr_Set_Vocabulary ( *cp );
    goto sendev;

case DVRC_GETWORDS: /* get voice-recognized words from user */
    DoGetWordsCommand();
    if ( lp->bSendEvent )
        goto sendev;
    goto finished;

case DVRC_GET: /* get options and parameters */
    switch ( cmd.nSetGetOpCode ) {
        case GET_FILEFORMATS:
            ep->nEvent = EV_GET;
            ep->cGetParm[0] = ( char ) NULL;
            ep->dwGetParm = FM_VOX;
            ep->result = RC_SUCCESS;
            break;
    } /* end of switch */
- 276 -

goto sendev;

/*******************************************/
default:
    /* send back event */
    ep->nEvent = RV_BADCOMMAND;
    ep->result = RC_BADCOMMAND;
    /* send the event */
    goto sendev;
}

} /* never get here */

sendev:
    /* send back event */
    *event = *ep;
    lp->bCmdInProgress = FALSE;
    /* fall through to ... */

finished:
    nVRetVal = ep->result;

done:
    if ( nVRetVal != RC_CMDNOTDONE )
        return ( nVRetVal );

} /* ld_Initialize - Verify that Dialogic driver is running, find */
/* out how many lines exist, initialize them. */
******************************************************************************/

int ld_Initialize( 
    const int ld_hw_irq, \ // HW IRQ for LD card
    const int ld_sw_int, \ //SW interrupt for LD card driver
    const long win_hInst, \ //hInst if using Windows
    const long win_hWnd, \ //hWnd if using Windows
    char *prompttext, \ // filename extension for prompt
    char *ifileext) \ // filename extension for indexed
                   \ prompt files
    
    {
        int aline_num; \ // active line num (0 to MAX_LINES-1 ) */
        int ifile_num; \ // indexed file number (0 to MAX_IFILE-1 ) */
        int nVRetVal; \ // Vowsys return value
        int nDialRetVal; \ // Dialogic return value

#if DO_SFXPARM
    DCS My_dcb; \ // dialog control block (global params) */
#endif

#if defined(_WIN32) \ // Windows */
    long dcb_DosPointer = 0L;
    short dcb_ProtectedSelector = 0;
    long dcb_ProtectedPointer = 0L;
#endif

    #endif
    #endif
    int GotEvent;
long far *address;

if ( Initialized ) {
    nRetVal = RC_ALREADYINITIALIZED;
    goto done;
}

// sanity-check a debug array
if ( ((sizeof (DiaEventNumToName)) / sizeof (char *)) != MAXTERM+1) {
    Debug ( 1, sprintf ( Debug_buf, "LD_TAPI/ld_Initialize: DiaEventNumToName sanity failure" ) );
    nRetVal = RC_INTERNAL;
    goto done;
}

strcpy ( promptext, PROMPT_FEXT );
strcpy ( ifileext, INDEX_FEXT );
nActualLines = 0;

/*
 * Test to see if the Dialogic driver is up and running.
 */
int_level = ld_sw_int; /* software interrupt level - must agree with driver */
nDialRetVal = getvctor ( );
if ( ( nDialRetVal == 0 ) || ( nDialRetVal != ld_sw_int ) ) {
    nRetVal = RC_NOLINE;
    goto done;
}

address = ( long far *) _dos_getvect ( ld_sw_int );
address = ( long far *) _dos_getvect ( ld_sw_int - 1 );
#if 0
nDialRetVal = isdrvact ( ld_sw_int );
if ( nDialRetVal != 1 ) {
    nRetVal = RC_NOLINE;
    goto done;
}
#endif

/*
 * Shut the system down. Ignore the return code; we don't care if the system is already initialised or not.
 */
nDialRetVal = stopsys ( );

#if DO_SETXPARM

    */
    * Initialize the Dialogic system.
    */
    clrdb ( &My_dcb );
    #if defined ( _W )
        alloc_DOS_mem ( sizeof ( My_dcb ),
                        &dcb_DosPointer,
                        &dcb_protectedSelector,
                        &dcb_ProtectedPointer );
    #endif
    if ( dcb_protectedSelector == 0 ) {
        nRetVal = RC_RAMFULL;
        goto done;


```c

- 278 -
}

memcpy(
    (void*) dcb.ProtectedPointer, /* dest */
    (void*) &dcb,/* source */
    sizeof ( My_dcb )
);

#endif

nDialRetVal = setxparm(
#if defined(B_W) /* Windows */
    dcb_DosPointer
#else
    &My_dcb
#endif
);

#if defined(B_W) /* Windows */
    free_DOS_mem ( &dcb_DosPointer, &dcb.ProtectedSelector, &dcb.ProtectedPointer);
#endif

switch ( nDialRetVal ) {
    case E_SUCCE: /* Function was successful. */
        break;
    case E_BADDL: /* D4x hardware error. */
        nRetVal = RC_NOLINK;
        goto done;
    default:      /* some unknown error */
        nRetVal = RC_INTERNALLINE;
        goto done;
}

#endif

nDialRetVal = startsys ( 1d_hw_irq, /* HW interrupt level */
    SM_EVENT, /* use Event mode */
    0, /* use buffers specified to driver */
    0, /* use buffers specified to driver */
    &nActualLines    // retvval -- number of lines available
    );

switch ( nDialRetVal ) {
    case E_SUCCE: /* Function was successful. */
        break;
    case E_FAILST: /* Board failed self test. */
    case E_SACT: /* System already active. */
    case E_SNACT: /* System not active. */
    case E_BADDL: /* D4x hardware error. */
    case E_BADDINT: /* Interrupt level not available. */
        nRetVal = RC_NOLINK;
        goto done;
    default:      /* some unknown error */
        nRetVal = RC_INTERNALLINE;
        goto done;
}

#if defined(B_W) /* Windows */
    /* allocate memory for Check_For_Event */
    alloc_DOS_mem(
        sizeof ( EVTBLK ),
        &cfe_DosPointer,
        &cfe.ProtectedSelector,
        &cfe.ProtectedPointer
    );
    if ( cfe.ProtectedSelector == 0 ) {
```

nRetVal = RC_RAMFULL;
goto done;
}
#endif

if ( nActualLines > MAX_LINES )
nActualLines = MAX_LINES;
if ( nActualLines > MAX_AL )
nActualLines = MAX_AL;

/* initialize array of active line states */
for ( aline_num = 0; aline_num < MAX_AL; aline_num++ ) {
    lp = &line[aline_num];
    lp->nLineNum = aline_num;
    lp->bConnected = FALSE;
    lp->bCmdInProgress = FALSE;
    lp->timeout = 0;
    lp->bUserIsOffHook = FALSE;
    lp->bCardIsOffHook = FALSE;
    lp->num_events = 0;
    lp->plist_count = 0;
    lp->open_file = (-1);
    for ( ifile_num = 0; ifile_num < MAX_IFILE; ifile_num++ )
        lp->ifile[ifile_num].ifhandle = (-1);
    lp->num_openifile = 0;
}

Initialized = TRUE;
nRetVal = RC_SUCCEEDED;

/*
 * See if we have voice recognition capabilities. They are
 * optional; not having them is not an error.
 */
vr_Initialize ( );

Check_For_Event ( &GotEvent );  /* !!! */
done:
    return ( nRetVal );
}

/**************************************************************/

int ld_Shutdown ( void )
{
    int nRetVal;  // Voyysey return value
    int nDialRetVal;  // Dialogic return value
    int aline_num;  /* active line num (0 to MAX_AL-1) */
    int ifile_num;  /* indexed file number (0 to MAX_IFILE-1) */
    boolean bNeedToWait;
    int nLineNum;

    if ( !Initialized ) {
        nRetVal = RC_NOTINITIALIZED;
        goto done;
    }
}
- 280 -

/* abort any command in progress on any line */
bNeedToWait = FALSE;
for ( nLineNum = 0; nLineNum < MAX_AL; nLineNum++ ) {
    lp = &line[nLineNum];
    if ( lp->bCmdInProgress ) {
        if ( lp->cmd.nCommand != DVR_DISCONNECT ) {
            lp->nCmdStage = SHUTDOWN_STAGE;
        }
        bNeedToWait = TRUE;
    }
}

if ( bNeedToWait ) {
    nRetVal = RC_BUSYLINE;
goto done;
}

/* fail if any line is connected */
for ( nLineNum = 0; nLineNum < MAX_AL; nLineNum++ ) {
    lp = &line[nLineNum];
    if ( lp->bConnected ) {
        nRetVal = RC_BUSYLINE;
goto done;
    }
}

Initialized = FALSE;

/* Shut down voice recognition (if any). */
vr_Shutdown ( );

/* Shut the system down. */
nDialRetVal = stopsys ( );

switch ( nDialRetVal ) {
    case E_SUCCEED: /* Function was successful. */
        break;
    case E_SNAKE: /* System not active. */
        break;
    default: /* some unknown error */
        nRetVal = RC_INTERNAL;
        goto done;
}

for ( aline_num = 0; aline_num < MAX_AL; aline_num++ ) {
    /* close any files left open (should never happen) */
    if ( line[aline_num].open_file >= 0 )
        close ( line[aline_num].open_file );
    /* close any indexed files that are open */
    for ( ifile_num = 0; ifile_num <
        line[aline_num].num_openifile; ifile_num++ ) {
        if ( line[aline_num].ifile[ifile_num].ifhandle >= 0 )
            close ( line[aline_num].ifile[ifile_num].ifhandle);
        line[aline_num].ifile[ifile_num].ifhandle = (-1);
    }
}

line[aline_num].num_openifile = 0;
#if defined(_WIN)
    /* Windows */
    /* free memory for Check_For_Event */
    free_DOS_mem ( &cfe_DosPointer, &cfe_ProtectedSelector,
                  &cfe_ProtectedPointer );
#endif

nVRetVal = RC_SUCCEEDED;
done:
    return ( nVRetVal );
}

int ld_Process_Event ( const UINT message, // Windows message number
                       const WPARAM wParam, // parameter
                       const LPARAM lParam // parameter
)
{
    int nVRetVal; // Voysys return value

    if ( !Initialized ) {
        nVRetVal = RC_NOTINITIALIZED;
        goto done;
    }

    Debug ( 1, sprintf ( Debug_buf, "LD_Dial/ld_Process_Event:
discard event 0x%04X", message ) );

    nVRetVal = RC_SUCCEEDED;

done:
    return ( nVRetVal );
}

/*********************/
1. Telephony server apparatus, for use with a plurality of telephony channels, comprising:
   a processor structure;
   channel control hardware coupled to said telephony channels and to said processor structure; and
   a memory structure having stored therein a plurality of channel programs each associated with a respective one of said telephony channels, multi-tasking operating system software instructions executable by said processor structure, database engine software instructions executable by said processor structure under a different task of said operating system for each of said telephony channels, and a database,
   said database engine software instructions including instructions which, when executing under a given task of said operating system, interpret the channel program associated with the telephony channel of said given task and perform both database and telephony operations in response to such channel program associated with the telephony channel of said given task, said database operations including at least one operation from the group consisting of reading data from and writing data to said database, and said telephony operations including at least one operation from the group consisting of speaking a predefined prompt onto the telephony channel associated with the given task, receiving and storing DTMF-encoded input from the telephony channel associated with the given task, and recording audio input from the telephony channel associated with the given task.

2. Apparatus according to claim 1, wherein said processor structure includes no more than one processor.
3. Apparatus according to claim 1, wherein said memory structure includes both semiconductor memory and rotating memory.

4. Apparatus according to claim 5, wherein the read-write data portion of the task associated with each particular one of said telephony channel includes the channel program associated with said particular telephony channel, and wherein all of said channel programs are the same.

5. Apparatus according to claim 1, wherein each of said tasks includes an executable portion and a read-write data portion, the read-write data portion being separate for each of said tasks, and the executable portion being common to all of said tasks.

6. Apparatus according to claim 1, wherein said database operations include all operations from said group consisting of reading data from and writing data to said database, and wherein said telephony operations include all operations from said group consisting of speaking a predefined prompt onto the telephony channel associated with the given task, receiving and storing DTMF-encoded numerical input from the telephony channel associated with the given task, and digitally recording audio input from the telephony channel associated with the given task.

7. Apparatus according to claim 1, wherein said telephony operations further include the operation of waiting for a ring signal from the telephony channel associated with the given task.
8. Apparatus according to claim 1, wherein said telephony operations further include the operation of answering a call from the telephony channel associated with the given task.

9. Apparatus according to claim 1, wherein said telephony operations further include the operation of hanging up a call on the telephony channel associated with the given task.

10. Apparatus according to claim 1, wherein said telephony operations further include the operation of dialing a call on the telephony channel associated with the given task.

11. Telephony server apparatus, for use with a first telephony channel, comprising:
    a processor structure;
    channel control hardware coupled to said first telephony channel; and
    a memory structure having stored therein a database and software instructions executable by said processor structure, said software instructions including:
    a database language sequencer;
    a database control module having a plurality of procedures callable by said database language sequencer for performing at least one of the database operations from the group consisting of reading selected data from and writing specified data to said database; and
    a telephony control module having a plurality of procedures callable by said database language sequencer for performing at least one of the telephony operations from the group consisting of speaking a predefined
prompt onto the first telephony channel, receiving and storing DTMF-encoded input from the first telephony channel, and recording audio input from the first telephony channel,
said database language sequencer calling said database control module procedures and said telephony control module procedures in a sequence defined by a program which satisfies predefined syntax rules of a predefined database language.

12. Apparatus according to claim 11, wherein said database language sequencer comprises:
said program; and
an interpreter which interprets said program to develop said sequence in which said database language sequencer calls said database control module procedures and said telephony control module procedures.

13. Apparatus according to claim 11, wherein said database language sequencer comprises a product produced by the method comprising the steps of:
providing said program; and
converting said program to software instructions executable by said processor structure.

14. Apparatus according to claim 11, wherein said processor structure includes no more than one processor.

15. Apparatus according to claim 11, wherein said memory structure includes both semiconductor memory and rotating memory.

16. Apparatus according to claim 11, wherein said database operations include all operations from said
group consisting of reading data from and writing data
to said database, and wherein said telephony operations
include all operations from said group consisting of
speaking a predefined prompt onto said first telephony
channel, receiving and storing DTMF-encoded numerical
input from said first telephony channel, and digitally
recording audio input from said first telephony channel.

17. Apparatus according to claim 11, wherein said
telephony operations further include the operation of
waiting for a ring signal from said first telephony
channel.

18. Apparatus according to claim 11, wherein said
telephony operations further include the operation of
answering a call from said first telephony channel.

19. Apparatus according to claim 11, wherein said
telephony operations further include the operation of
hanging up a call on said first telephony channel.

20. Apparatus according to claim 11, wherein said
telephony operations further include the operation of
dialing a call on said first telephony channel.

21. Apparatus according to claim 11, for use further
with a second telephony channel, wherein said database
language sequencer, said database control module and
said telephony control module comprise first
instantiations of said database language sequencer, said
database control module and said telephony control
module, respectively, and wherein said memory structure
further has stored therein:
a second instantiation of each of said database
language sequencer, said database control module and
said telephony control module,
the procedures of said second instantiation of said
telephony control module speaking a predefined prompt
onto said second channel, receiving and storing DTMF-
encoded numerical input from said second channel, and
digitally recording audio input from said second
channel.

22. Apparatus according to claim 21,
wherein said first and second instantiations of said
database language sequencer share a common set of
software instructions and have different instance data,
wherein said first and second instantiations of said
database control module share a common set of software
instructions and have different instance data,
and wherein said first and second instantiations of
said telephony control module share a common set of
software instructions and have different instance data,
the instance data for said first instantiation of said
telephony control module identifying said first
telephony channel and the instance data for said second
instantiation of said telephony control module
identifying said second telephony channel.

23. Apparatus according to claim 22,
wherein the common set of software instructions
shared by said first and second instantiations of said
database language sequencer includes an interpreter
which interprets a database language program to develop
said sequence in which said database language sequencer
calls said database control module procedures and said
telephony control module procedures,
wherein the instance data of said first instantiation of said database language sequencer identifies a first program as the database language program to interpret,

and wherein the instance data of said second instantiation of said database language sequencer identifies a second program as the database language program to interpret.

24. Apparatus according to claim 22, wherein said software instructions stored in said memory structure further include:

a channel server common to both said first and second instantiations of said telephony control module, said channel server performing said telephony operations on said first telephony channel in response to communications from said first instantiation of said telephony control module, and performing said telephony operations on said second telephony channel in response to communications from said second instantiation of said telephony control module.
FIG. 1
FIG. 3

FIG. 4
### INTERNATIONAL SEARCH REPORT

**International application No.**

PCT/US95/15537

#### A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : HO4M 1/57, 1/64, 1/50  
US CL : 379/67  
According to International Patent Classification (IPC) or to both national classification and IPC

#### B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

| U.S. | 379/67, 88, 89, 201, 207; 395/600; 364/300; 273/439 |

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic database consulted during the international search (name of database and, where practicable, search terms used)

APS

#### C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>US, A, 5,354,069 (GUTTMAN ET AL) 11 OCTOBER 1994. See Figures 4A, 4B, 4C.</td>
<td>1, 6-10</td>
</tr>
<tr>
<td>A</td>
<td>US, A, 5,113,430 (RICHARDSON, JR. ET AL) 12 MAY 1992. See Figures 2, 3; Column 7 line 15 - Column 8 line 37.</td>
<td>1-24</td>
</tr>
<tr>
<td>A</td>
<td>US, A, 4,695,977 (HANSEN ET AL) 22 SEPTEMBER 1987. See Figure 10; Column 2, line 50 - Column 4, line 15.</td>
<td>1-24</td>
</tr>
<tr>
<td>A</td>
<td>US, A, 5,255,305 (SATTAR) 19 OCTOBER 1993. See Figures 1, 2A; Column 7.</td>
<td>1-24</td>
</tr>
</tbody>
</table>

Further documents are listed in the continuation of Box C. See patent family annex.

| * | Special categories of cited documents:
| **A** | document defining the general state of the art which is not considered to be part of particular relevance
| **E** | earlier document published on or after the international filing date
| **L** | document which may throw doubt on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
| **O** | document referring to an oral disclosure, use, exhibition or other means
| **P** | document published prior to the international filing date but later than the priority date claimed
| **T** | later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
| **X** | document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
| **Y** | document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
| **Z** | document member of the same patent family

Date of the actual completion of the international search: 25 FEBRUARY 1996

Date of mailing of the international search report: 05 MAR 1996

Name and mailing address of the ISA/US

Commissioner of Patents and Trademarks

Box PCT  
Washington, D.C. 20231

Authorized officer: PARAG DHARIA

Facsimile No. (703) 308-5401  
Telephone No. (703) 308-5456

Form PCT/ISA/210 (second sheet) (July 1992)