

ENHANCEMENTS TO KAPPA, AN OBJECT
ORIENTED EXPERT SYSTEM SHELL

By

ROHINTON NOSHIR MISTRY

Bachelor of Science

Osmania University

Hyderabad, India


1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1992

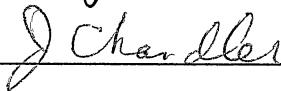
Atsui
1992
1-16782

ENHANCEMENTS TO KAPPA, AN OBJECT
ORIENTED EXPERT SYSTEM SHELL

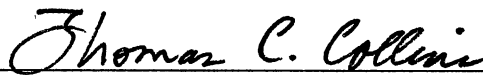
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Dr. Blayne E. Mayfield for his encouragement and advice in successfully completing this thesis. Special thanks go to Dr. Charles M. Bacon who helped and encouraged me with his valuable suggestions, advice and support. Many thanks also go to Dr. John P. Chandler for serving on my graduate committee and his sincere advice throughout my graduate program.

My parents, Noshir and Rashna Mistry, encouraged and supported me all the way, helping me to keep the end goal constantly in sight. My special thanks go to my parents, family and friends in their love and encouragement which helped me achieve my goal.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. OVERVIEW	3
Object Oriented Concepts	3
Object Oriented Concepts in KAPPA	6
Expert Systems	7
From the Old to the New	8
Characteristics of an Expert System	10
Knowledge Representation	11
The Inference Engine	12
The User Interface	13
KAPPA as an Expert System Shell	14
Windows	15
Windows and KAPPA	18
III. TOOLS USED FOR ENHANCEMENTS TO KAPPA	21
The Tools	21
The Method	24
IV. THE ENHANCEMENTS TO KAPPA	27
Existing String and File Functions	27
The New Functionality	28
Salient Features of Some String Functions	30
Salient Features of Some File Functions	31
Salient Features of Other Miscellaneous Functions	33
Other Miscellaneous Enhancements to KAPPA	34
V. COMPARISON AND ANALYSIS OF ENHANCEMENTS IN KAPPA ..	36
Comparison of Old and New Features	36
Consequences of the New Functionality	41

Chapter	Page
Comparison and Analysis of User Defined Functions	
With and Without the New Functionality	42
Working with Old KAL Functions	42
Calling External Executable Files	48
Interfacing with C into KAPPA	51
Comparison with Newer Versions of KAPPA	52
String Functions	52
File Functions	54
Other Enhancements	55
 VI. SUMMARY AND CONCLUSIONS	 56
 BIBLIOGRAPHY	 60
 APPENDIXES	 63
APPENDIX A - THE KAPPA.C PROGRAM FOR INITIATING KAPPA AND REGISTERING FUNCTIONS	 64
APPENDIX B - ALPHABETICAL REFERENCE FOR ALL ENHANCED FUNCTIONS	 68

LIST OF FIGURES

Figure	Page
1. Class-Instance Hierarchy	4
2. A Basic Expert System	10
3. The KAPPA Opening Screen	19
4. A Sample KAPPA Session Layout Screen	19
5. Makefile for Compiling and Linking	24
6. Example Data Files	43
7. Reading Example1.TXT	44
8. Reading Example2.TXT	45
9. Comparison of Code Sizes	47
10. Comparison of Execution Times	48
11. C and KAL Functions to Convert a String to Uppercase	50

CHAPTER I

INTRODUCTION

KAPPA is an object oriented expert system shell or tool, developed by IntelliCorp Inc. [1]. It provides a powerful environment for creating, developing and delivering applications and knowledge-based expert systems. It is PC-based, works under Microsoft Windows [4] and takes advantage of various features provided in the Windows environment. The specific version of KAPPA used in this thesis is KAPPA 1.1X, and any reference to KAPPA will indicate this version unless specified otherwise.

The language used in KAPPA is KAL, the KAPPA Application Language, which consists of many different functions to perform a wide variety of knowledge, string, file, list and user interface manipulations. One of the weaker areas in KAPPA is its lack of good string and text file handling functions. Though able to access and manipulate a wide variety of knowledge bases such as database and spreadsheet files, it has limited abilities in handling knowledge bases existing in the ASCII text format. It also has limited ability to manipulate these text strings, once they are read into KAPPA.

This thesis deals with new functionality added to KAPPA in the areas of string and text file handling to further enhance its capabilities. The functionality is added using the C interface provided by KAPPA. The enhanced functions are written in C and Windows, and are then linked with KAPPA libraries provided, to form a new executable

file. Further, a thorough analysis is performed on these enhancements to show the effectiveness, uses and advantages after the implementation.

Principles of object oriented programming, expert systems, Windows and KAPPA are explained in Chapter II. The enhancements are described in detail in Chapter IV and the appendices, while the tools and methods used, are described in Chapter III. Chapter V describes different aspects of the comparison and analysis performed on the enhanced functions to show their usefulness.

This thesis shows the weaknesses of KAPPA before the enhancements were made, in terms of string and file handling, and the overall effectiveness of the new functions added to it. It shows how, where and when to apply these new functions and the advantages and disadvantages of adding these functions.

Some sample application functions are taken from an expert system project for configuring industrial filtration systems and are analyzed to show the differences in incorporating the new functions and writing "workarounds". The "workarounds" are functions written by means other than incorporating the enhanced functions directly. This may entail lengthier code in KAL or writing and calling functions defined in other languages. The "workarounds" therefore take considerably more development effort in terms of ease, size, time and effectiveness. These disadvantages are discussed in Chapter V.

Chapter VI contains a summary and offers conclusions. It discusses other weaknesses of KAPPA and areas where further enhancements can be made.

CHAPTER II

OVERVIEW

Object Oriented Concepts

The concepts of object oriented programming were first introduced in Simula, a goal-based simulation language wherein the fundamental ideas of objects, classes and messages were used [18, 19, 20]. Smalltalk soon followed, building upon the concept of an object class [18, 21, 22, 23].

An object oriented program is a collection of encapsulated data structures called objects. An object contains two basic types of information; one describing the object and another specifying what it can do [1, 18]. Instead of manipulating or viewing the contents of an object directly, messages giving instructions, are sent to objects and the object can select a method by which to react to the message. With such encapsulation, possibilities of multiple object instantiation, behavioral sharing through inheritance and structuring of resources in applications can be done, as will be shown.

Objects are generally declared as classes and instances. A class is a much more general classification than an instance. A class represents a collection of items which have certain properties in common. This class may be further divided into sub-classes which come under their parents in a broad category, but have unique properties of their own which differentiate them from other siblings of the same parent class. At the lowest

level are the instances of the classes or sub-classes. These are like individual examples of their parent classes. Taking as an example courses offered within the Graduate College as a class, we may specify sub-classes such as Engineering, Arts&Sciences and Business. These may be further subdivided into Civil, Mechanical and Electrical Engineering sub-classes; Math, Physics and Chemistry Science classes; and Finance, Marketing and Accounting Business classes. Individual examples or instances of these sub-classes could be the different courses offered such as EE 5010, EE 5050 and EE 5290. These relationships are illustrated in Figure 1.

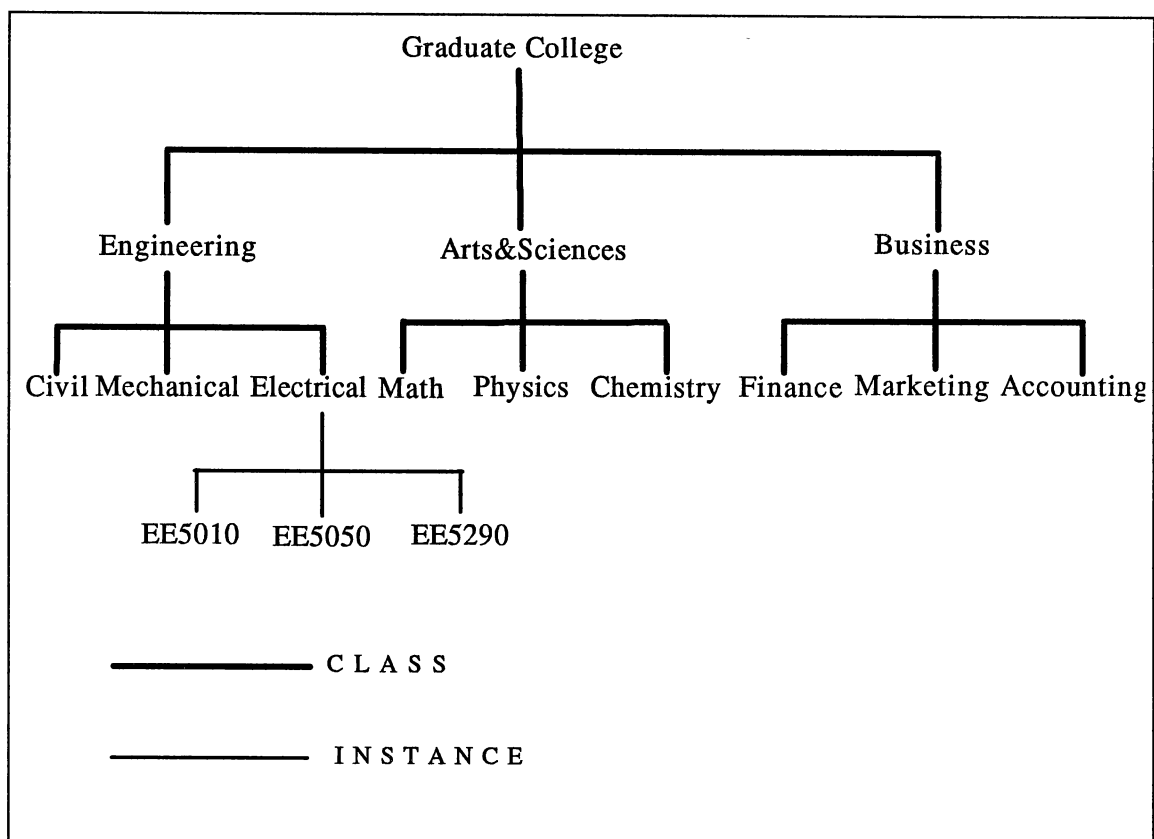


Figure 1. Class-Instance Hierarchy

The classes and instances thus form an hierarchical tree structure or Directed Acyclic Graph (DAG). The classes and instances can store all the information related to them within its structure. Information and properties stored in classes higher up in the hierarchy is also inherited by their sub-classes and instances. Thus if the Graduate College in the above example contains Masters students, all students here for a particular course will be Masters students. Similarly, if Engineering students get MS degrees, all the individual students graduating with Engineering courses will get MS degrees.

All communication between objects is done by some form of message passing. When an object receives a certain message, it can decide whether action has to be taken or not, and what action to take if any. This supports data abstraction, where the object uses its own methods to modify itself and the message sender makes no assumptions about the internal representations of these objects. This concept is different from the classical concept where the programs and data were always stored separately with the programs acting upon the data [18]. Objects consist of both data and programs (called methods), stored together in an encapsulated form.

The encapsulation of data and methods offer many advantages [31]. Complex relationships between data elements which are difficult to capture in the abstract, can often be more accurately modelled by the structure of objects and messages that determine how the objects interact with one another. Interdependence among objects is strictly limited by the messages that are sent and received. A change within one object will not affect the behavior of another object, thus the maintenance of an object system will not be nearly as tiresome as it can be with larger systems that are not object oriented. Larger systems can thus be built using interrelated but independent objects.

Object oriented programming has lately become very popular in different areas of computer science including expert systems, artificial intelligence, graphics, systems programming and databases. The above described concepts have been used to write expert system shells and databases as they provide a more realistic representation of related code and data being together. KAPPA is a good example of an expert system shell which provides the user with object oriented tools in addition to traditional rule based reasoning, to build applications.

Object Oriented Concepts in KAPPA

In KAPPA, classes and instances are defined and linked together in a hierarchy to represent the relations between the objects. A graphical representation of the hierarchy is displayed by the object browser. The graphical representation, much like the one displayed in Figure 1, is displayed by the browser with solid lines denoting classes, while dashed lines denote instances. The user interface is designed so that a pointing device such as a mouse can be used to click on any class or instance name and modify, create or delete that object.

Slots can be defined within classes or instances and can contain properties or information related to that particular object. Slots are variables defined inside objects and, in addition to defining them explicitly, they can also inherit values of similar slots higher up in the object hierarchy. Slots represent the lowest level where specific information relating to the broader classes or instances is stored. Numbers, text, boolean or other object values can be stored within slots. They can be set to contain either a single value or multiple values (lists). Certain slot options such as the maximum and

minimum values for numeric slots and the allowable values in a slot can also be set. To monitor the slots when their values change, different methods can be defined to be activated before, after or when needed.

Each action that an object can carry out is represented by a method [1, 17]. These define the behavior of the objects and slots within objects. Methods determine when and how the values in these slots change with the method of activation specified by sending a message. When an object receives a certain message that corresponds to one of its methods, the method is activated. Many objects can have the same method, thus responding to the same message. Methods can also be inherited, the same as slots.

By defining classes, instances, slots and methods as shown above, the traditional rule based reasoning can be augmented, as this produces a more intuitive representation and enhances speed and modularity in a knowledge base of relatively independent modules.

Expert Systems

Expert system shells are development platforms used to build expert systems. The shells provide an easier, generally menu driven environment for creating expert systems. Expert system programs have certain distinguishing features and are used mainly for knowledge manipulation. Thus expert systems are sometimes also called knowledge systems. They are mainly to use and manipulate large amounts of data and expertise stored within them.

An expert system tries to emulate an expert in the use of his knowledge. The knowledge required by a human expert in a given field is stored as the knowledge base

within the expert system. The rules governing the way the human expert manipulates this knowledge to give him the required results, is emulated in the expert system by traditional rule based reasoning or, in object oriented systems, by using methods which are applied to the various objects.

From the Old to the New

Alan Turing [27] is credited as the first to see clearly the possibilities of thinking machines, as early as 1950. John McCarthy and Marvin Minsky [27], in a conference at Dartmouth in 1956, first coined the term "artificial intelligence". DENDRAL, in 1968, is one of the first successful expert systems [27]. It uses three steps to identify the structure of a parent compound from an input formula, mass spectrometer and magnetic resonance data. It is a landmark program, as it was the first time someone sat down with a human expert to determine the heuristics, as well as the constraints involved in solving a complex problem. Some of the other expert systems designed in the 70's were PROSPECTOR, MYCIN and XCON (also known as R1) [24]. Other popular expert systems were INTERNIST, CADUCEUS, HEARSAY, and PUFF [30]. Some of the popular applications of expert systems are in areas including medicine, chemistry, geography, natural language processing, psychology, space, finance, communications, planning, estimating costs, problem solving, training, inventory control, customer support, evaluation, diagnosis and military systems [29, 30].

Block structured languages like Pascal are procedural and have few mechanisms to describe a closed world of facts, hence list processing languages such as LISP and logic processing languages such as PROLOG were preferred as expert system writing

tools [27]. LISP computers were difficult to integrate with conventional systems and lacked file security, hence a tendency towards C based systems then took place [32]. As the programming skills required and the time spent in writing expert systems is considerable when using these languages, the expert system shells available are now being sought as expert system writing tools. These shells provide a wide range of features including an inference engine to manage the rules and deduce the program flow, hence requiring much less time and effort spent in building expert systems. Recently, such shells have become available for use on micro computers. These include Expert-Ease, EXSYS, Level 5, Personal Consultant, NEXPERT, PEx, KEE, ExperTax, EXSEL, 1st Class Fusion, Guru, Level5, VP-Expert and KAPPA [26, 32, 33].

The expert system shells or tools can be divided into five general categories, each using a distinct method of reasoning [26, 36]: induction, simple rule-based, structured rule-based, object oriented hybrid and logical tools. The induction tools learn from a large number of examples from which rules are automatically formed. They are friendly but weak and inflexible. Simple rule-based tools use simple if-then rules to represent knowledge. The structured rule-based tools, on the other hand, can be arranged into hierarchies and one set of rules can inherit information acquired when other sets of rules are examined. The object oriented hybrid tools use object oriented techniques in addition to rules. Lastly, logical tools use horn clauses and resolution strategies derived from Predicate Calculus. They are powerful when dealing with very complex problems that lend themselves to rigorous logical analysis.

Characteristics of an Expert System

An expert system has three functional components [28]. It contains a knowledge base which stores the information, an inference engine that reasons with this information, and an interface that makes the software understandable to the user. The expert system can be considered an artificial intelligence system as it reasons with the stored information, can explain this reasoning to the user and can handle uncertain data and conditions. It can be visualized as in Figure 2.

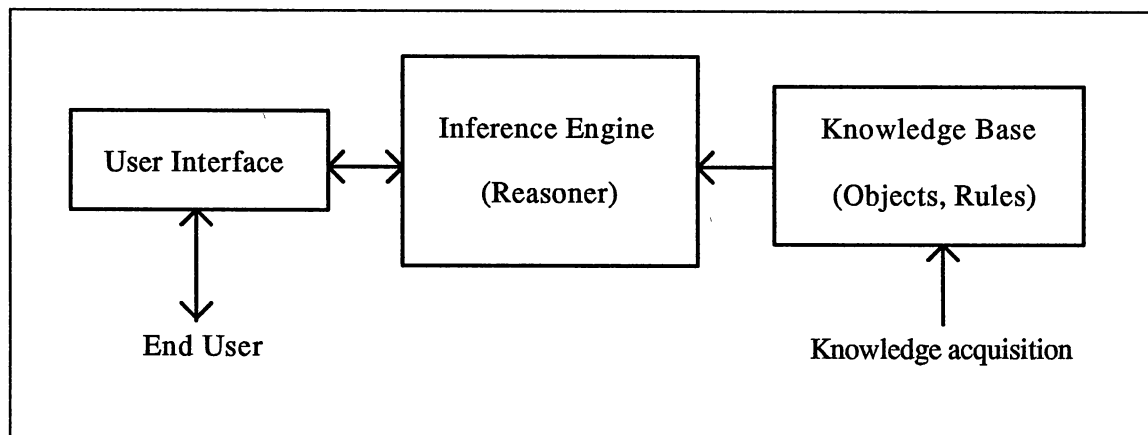


Figure 2. A Basic Expert System

Some characteristics of an expert system, according to Paul Siegel, are as follows [28]. It is subtle, as it can make the user learn and use it effortlessly. It is flexible in terms of the information stored on disks, which can be easily modified and can accommodate changes in technology and business organization. It is efficient and

effective, as once built, many people may use it effectively without formal training. It is empowering, as workers do not feel like trainees, and when anything goes wrong, they can fix the problem themselves by using the expert system. Lastly, it is compatible with the training professional, as the person, called a knowledge engineer, who develops the expert system, can use expert system shells to build and test expert systems so that the final system gives valid advice.

In general, the characteristics of an expert system [26, 37] are its ability to solve difficult problems as well as or better than humans, reason heuristically using the rules of thumb (ie. knowledge drawn from experience), interact with humans in appropriate ways including the use of natural language, manipulate and reason about symbolic descriptions, function with erroneous data and uncertain judgmental rules, contemplate multiple competing hypotheses simultaneously, explain why certain questions are asked and justify their conclusions.

Knowledge Representation

There exist a number of techniques for representing knowledge in expert systems [26, 35]. Production rules are two-part, if-then rules whose antecedent represents some pattern to be matched and a consequent that specifies an action to be taken when data matches the pattern. Semantic networks represent abstract relations among objects and may be represented graphically by a network of nodes and links where the nodes represent the objects and links represent their relations. Frames, which are similar to semantic networks, represent objects by containing relations to other objects which can also be inherited, unlike nodes. First-order logic (FOL) is a formal way of representing

logical propositions and relations between propositions where rules of logic can be easily applied to these representations to derive facts that follow from these propositions.

Recent expert systems use a combination of these and other knowledge representations.

The Inference Engine

An inference engine is a sub-routine or function that can deduce and infer new facts from those already existing. Rules or heuristics enable the derivation of new facts based on what is already known. Two reasoning mechanisms are commonly used in inference engines, alone or in combination. They are forward chaining and backward chaining.

The forward chaining or data-driven inference attempts to reason forward from the facts to a solution. Forward chaining occurs when a premise is stated and a conclusion has to be obtained from it.

In backward chaining or goal seeking, the goal or conclusion has to be established, and the system works backward from a hypothetical solution to find evidence supporting the solution by formulating and testing intermediate hypotheses.

Taking an if-then rule, the forward chaining looks for conditions matching the if-part (antecedent) of the rule and applying the then-part (consequent) of the rule, whereas the backward chaining matches the goal with the then-part of the rule and gains its evidence from the if-part. Partial or uncertain premises can be incorporated by including certainty factors (CF) which state the reliability of its conclusions.

Once a system becomes large, more than one rule can "trigger" (ie. become ready to fire) at the same time [27]. To overcome this sort of difficulty and enforce

consistency, the inference engine manages the rules according to conflict resolution strategies. An agenda of the potential actions awaiting execution is maintained and a scheduler controls and decides what to do next. The scheduler has to search for and apply the required rule.

A heuristic search uses different methods to narrow down the search space by intelligent pruning. Directing the search flow efficiently is a primary function of the inference engine. A number of search strategies exist including depth-first, breadth-first and hill-climbing. Of these, the first two are not heuristic. The depth-first search exhaustively searches all possible rules related to the rule condition items it last comes across, before moving on to other rules, whereas a breadth-first search exhaustively searches all possible rules related to the first item before searching other rules related to the other rule condition items. The hill-climbing strategy, on the other hand uses local information to decide which is the best path, which may vary from application to application.

The User Interface

The user interface varies a lot from system to system, depending a lot on the developing tools used. Ultimately, the systems developed have to be used by an end user [27]. Thus, it is critical that the user interface be well designed and as natural as possible to an inexperienced user, as well as an advanced user. Features like on-line help should be available and easily accessible.

A good interface should be able to prompt the user well, and accept answers in different ways. A user may select one, none or more than one item and may use the keyboard or a pointing device such as a mouse to enter his or her input.

If multiple answers are found and presented, some explanation of the rankings and probability factors, showing the probability or certainty of an event occurring, should also be provided. If graphs or charts can produce better effects, the system should have the ability to produce and use these.

The system should, at the very least, be menu driven and provide flexible screen management facilities. The user should also be able to query the system easily to find out why certain questions were asked and what reasoning process was used.

The system interface, which may function in a library environment would also require hooks to existing databases and other languages. If the expert system itself could be compiled into an executable, it would run faster and could be reproduced and ported to other machines, without the need of the shell or compiler being ported with it.

The interface provided for developing the expert system itself should be as flexible as possible. Full screen creation and editing facilities would be convenient to modify the knowledge base, rules, methods and functions. Good debugging and tracing facilities and a cross-referencing facility to find affected objects would also be useful.

KAPPA as an Expert System Shell

KAPPA is an object oriented hybrid expert system shell which runs under the Windows environment. Knowledge representation in KAPPA is in the form of classes, instances and slots, as already discussed. In addition, rules can also be used to represent

knowledge. The rules can be used in forward or backward chaining. A goal must be specified if chaining backward, but there is no need to specify goals when chaining forward. When the goal condition is satisfied, or when all related rules have been fired, the chaining process stops. Rules can also be categorized in KAPPA and different sets of rules can be used separately, increasing the modularity [1].

Depth-first, breadth-first, selective and best-first precedence and recency order of rules are all possible [1]. A priority can also be attached to each rule, which is then used for conflict resolution.

Independent functions can also be written in other languages and called from KAPPA. Functions are available to access knowledge bases in ASCII, dBase or Lotus 123 formats. The application development language provided is KAL [1, 2], the KAPPA Application Language, which is similar in many ways to the C programming language.

KAPPA has a very good and powerful graphical user interface, making use of the various features of Windows, including buttons, bitmaps, dialog boxes, sliders, meters and drawing tools to plot graphs [1, 2]. This interface uses both the keyboard and the mouse, making it easy for a novice or other inexperienced user to learn, adapt and use this environment. However, it has very little on-line help, no hypertext ability and applications developed on it cannot be made into stand-alone executables.

Windows

The Microsoft Windows environment, version 3.0 [4] provides the user with many features. It is essentially a multi-programming environment for the single user. It provides the user with a graphical user interface (GUI), in which multiple programs can

be displayed within windows and provides a more intuitive and efficient environment to work with.

Windows can be called up and run in three modes: 386 enhanced, standard and real. The 386 enhanced mode runs on a computer with the Intel 80386 or higher processor with 640 kilobytes of conventional memory and 1024 kilobytes or more of extended memory. Running in this mode provides increased control over non-Windows applications and provides the user with many advanced features. The standard mode can be used on a personal computer with the Intel 80286 or higher processor and 640 kilobytes of conventional memory plus 256 kilobytes or more of extended memory. In the real mode, it does not use extended or expanded memory and can run on a personal computer with an Intel 8086 or higher processor with 640 kilobytes of conventional memory.

The primary advantage of Windows is its ability to run more than one application at a time. Each Windows application (and non-Windows application in the 386 mode) can be run in a separate window by itself. A window is a rectangular portion of the screen dedicated to each different application. These windows can be sized, moved, iconified or maximized for effect.

Windows uses a non-preemptive, processor sharing mechanism where each application is given equal opportunity to run. Windows does not preempt an application to give another application a turn at running; each application itself hands over control to Windows when waiting for input or at regular intervals. Windows handles all functions by message passing.

The graphical environment is event-driven. When an event such as a mouse click occurs or a key is depressed, the GUI traps the event and sends a message to the appropriate window. The application then handles the message as required and hands back control to Windows. Priorities for applications running in the foreground (current) and those in the background are different and can be set.

The graphical user interface is not only much easier to use, [34] but also provides the programmer with a rich application programming interface (API) and device independence. Different graphics drivers such as CGA, EGA, VGA and Hercules are already built into Windows, thus giving an application developer device independent functions to work with, without the worry of different types of displays, communication software or printers. Thus, a general type of application can be written which will work with different displays and use any type of printer supported by Windows.

Windows also provides the ability to cut and paste data between different programs and applications. This provides an easy way of transferring information between different applications. It can also create "hot links" (sharing of data between two Windows applications by message passing, while the applications are running) with the Windows' Dynamic Data Exchange (DDE) protocol which makes it easy to integrate your application with other Windows programs.

Windows provides a vast number of functions of its own to create window based user applications. Menus, dialog boxes and different graphical images are easy to create. Dialog boxes allow the end user to select items displayed in a list like form with buttons to select, or display a variety of buttons for the user to choose from. Dialog boxes can contain buttons, list boxes, edit boxes, text boxes, combo boxes, scroll bars and other

graphics which can be used to input and display data [8, 12]. The Software Development Kit (SDK) provides many libraries, functions and utilities to build and debug Windows applications. Other software, such as Borland C++ [9, 10, 11, 12], can also be used to write and debug Windows applications.

All these features make Windows an ideal environment to work in, and applications designed to work in it can take advantage of these features to give the end user a more intuitive and efficient work environment.

Windows and KAPPA

KAPPA makes use of all the above Windows features to provide the user with an easy-to-use graphical user interface to work with. The KAPPA opening screen contains three open windows by default. The main window contains icons for the seven principle windows available in KAPPA [1]. Of these, the first two are already open, the Object Browser and the Knowledge Tools. Figure 3 shows the opening screen in KAPPA.

The Object Browser shows a graphical representation of the classes and instances hierarchy present in the application. The solid lines show classes while the dashed lines indicate instances. The Root, which is the basic class, always exists and so does the Image class, which contains various graphical images and their settings. A Global instance also exists at start-up. The image can be scaled or printed. The display of sub-classes or instances can be suppressed, in which case, the parent class is shown within a box. The mouse can be used to click on any class or instance and modify, delete or add to it.

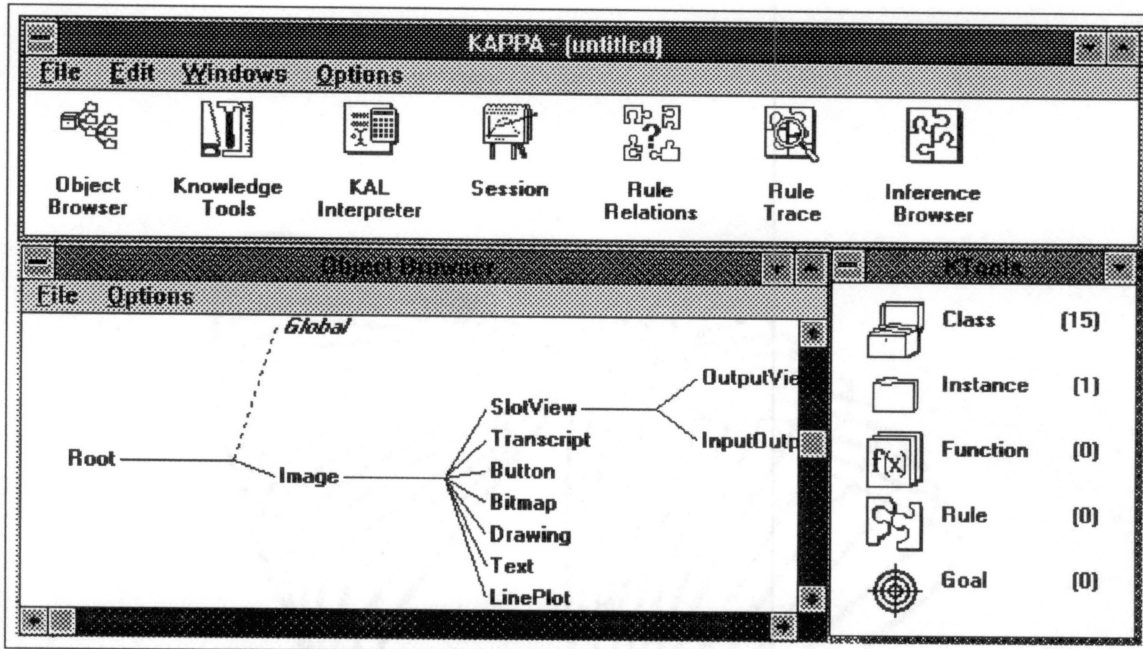


Figure 3. The KAPPA Opening Screen

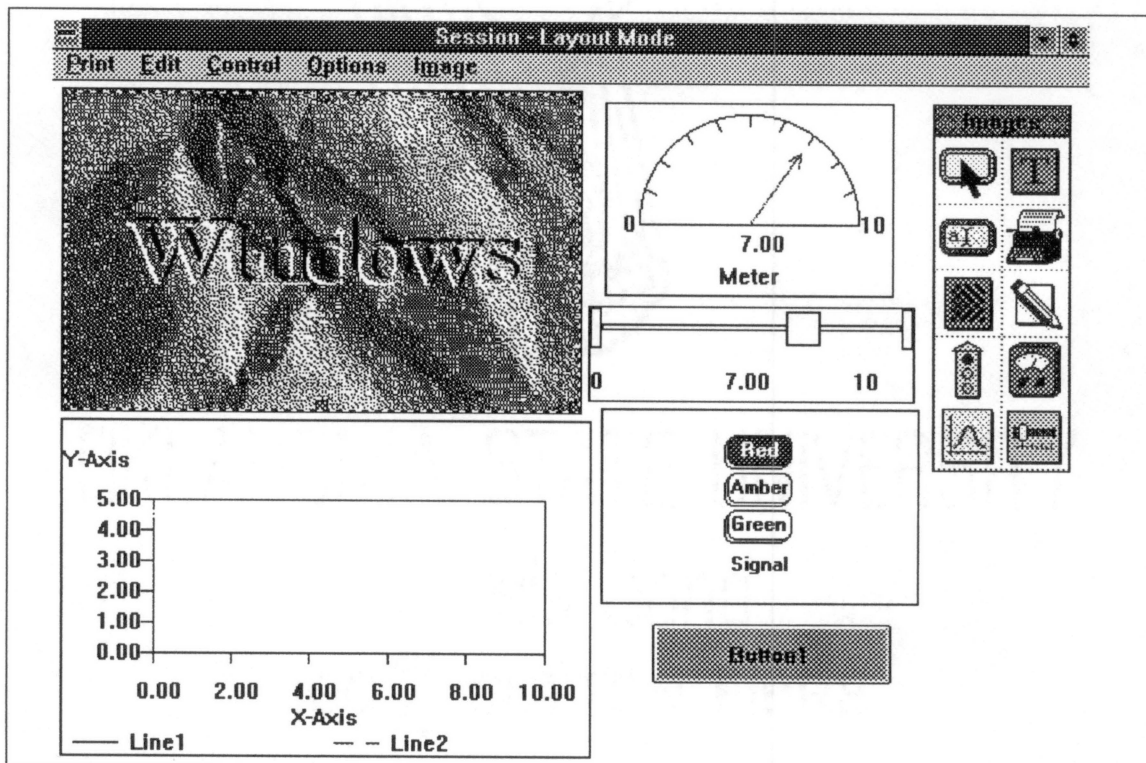


Figure 4. A Sample KAPPA Session Layout Screen

The Knowledge Tools provide editors for creating, modifying or deleting classes, instances, functions, rules and goals. The KAL interpreter can be used to type in commands that are executed immediately. It can be used to display and edit slot values, assert items into an agenda for chaining, or directly run functions. Everything that can be done using menus, editors and the mouse can be done in the KAL window.

The Session window contains the end user interface where the developed applications run. The graphical user interface can be used to advantage here to display buttons, meters, sliders, transcripts, graphs, bitmaps and various menus for the user to work with. A sample KAPPA Session Window screen in the layout mode, where the different images can be created and positioned, is displayed in Figure 4.

The Rule Relations window shows the if and then dependencies of any rule specified. The user can click on the rules displayed and edit or show the dependencies for any of the rules. A Rule Trace window traces the progress of each rule and the values contained in slots within them while the chaining process takes place. Break points can also be set to stop the chaining process. Finally the Inference Browser graphically displays the chaining relations among rules. It can show the object:slot pair that initiated the inference and the subsequent chaining process that took place.

CHAPTER III

TOOLS USED FOR ENHANCEMENTS TO KAPPA

The Tools

The KAPPA C interface [3] is required to make any changes to the KAPPA executable file. Essentially, the new routines are written in C, compiled, and linked with KAPPA C functions provided to produce one KAPPA executable file.

Although the KAL functionality of KAPPA can be extended from within by defining new user-defined functions, using registered C functions which are explained later in the chapter, by interfacing them has the advantage of speed and memory enhancements. The KAL functions are interpreted while registered C functions are binary as they are incorporated into the KAPPA executable file and thus are processed faster. The registered C functions also make better use of memory than KAL user-defined functions in KAPPA.

To interface the C code into KAPPA, the Microsoft Windows Software Development Kit (SDK) [5, 6, 7, 8] and a C compiler such as Microsoft C (recommended by KAPPA) or Borland C++, which has the ability to compile Windows executable files, must be available.

The Borland C++ compiler [9, 10, 11, 12] was used here as it provided many Windows capabilities with an easy to use editor, compiler and debugger. Using Borland

C++, the SDK was not necessary except for two library files required -- libw.lib and mlibcew.lib -- to make it compatible to the KAPPA libraries provided.

In addition to a C compiler and the SDK libraries, the KAPPA C library package is also required. The package consists of kappa.c, kappa.def, kappa.h, kappa.mak, kappa.res, kappac1.lib and kappac2.lib.

The kappac1.lib and kappac2.lib files contain all the necessary KAPPA object files necessary to make KAPPA. It contains all the routines, functions and capabilities of KAPPA.

The kappa.c file, when made into an object file, is used to initialize and register all the functions and routines of the C interface. The kappa.c file has to be modified to register all added C functions to be used with KAL. The kappa.h file is included in this file to use the Register_Function to register the C functions. The enhanced C functions file can also be included in this file.

Kappa.h is the KAPPA include file containing all the macros and declarations for knowledge manipulation. It contains all the in-built KAPPA functions available in KAL which can be made use of when interfacing C functions. It also contains special functions to change variables to and from C or KAPPA types. Names of menus, dialog boxes and other Window handles and object IDs required when manipulating KAPPA windows are also present in this file.

The kappa.res file is the resources file containing all the Window resources used by KAPPA, such as menus, dialog boxes, accelerator keys, bitmaps, icons, cursors and strings used in them. It is possible to use a resource editor such as the SDK tools [8] or Borland's Whitewater Resource Toolkit [12] to change these menus, dialog boxes or

other resources. These editors are menu driven, making it easy to modify or add resources to Window files.

The kappa.def file contains all the Microsoft Windows specific definitions required to make a Windows executable file. This module definition file [6] defines the application's contents and system requirements for the Windows linking program. It contains one or more of the code and data segment attributes, a one-line description of the module, the type of header, imported and exported functions, heap-size, stack-size, module name, additional code segments and an old-style executable stub file-name.

Some modifications must be made to the kappa.def file if procedures that are called by Windows are declared. The new procedure names have to be appended to this file and a corresponding ordinal number be defined. The code and data attributes can be set to preload. If the automatic data segment comprising of the heap size, stack size and the static code exceeds the 64K limit, the heap size or stack size has to be lowered correspondingly.

Lastly, the kappa.mak file is a make-file to combine, compile and link the various modules necessary in making the KAPPA executable file. The make-file is geared towards compiling and linking with Microsoft C, and would require changes to make it use the Borland compiler and linker instead, as shown in Figure 5.

This file can be renamed as makefile. which is the name of the default make-file for Borland's C++. A make command given directly at the DOS prompt would be sufficient to run it. The auto dependency check ensures that the compilation takes place when any included files are updated, such as the enhanced.c file which is included within kappa.c.

```

# Makefile.

# To do an automatic dependency check on all included files
.autodepend

# To obtain maximum memory to work with; makefile. will be swapped from memory
.swap

# Macro defining the default settings for compilation to an .obj file
cp=bcc -c -mm -W -O

# The goal
all: kappa.exe

# Compile kappa.c to obtain kappa.obj if required
kappa.obj: kappa.c
$(cp) -zC_RES kappa.c

# Link and compile resources to obtain kappa.exe, if required
kappa.exe: kappa.obj
tlink /Tw /n /x /A=16 kappa,,kappac1 kappac2 libw mlibcew,kappa.def
rc -t kappa.res

```

Figure 5. Makefile for Compiling and Linking

As KAPPA uses the Medium size memory model, the object code created has to follow the same memory size conventions, specially when making calls to the standard C library memory allocation routines.

The Method

KAPPA has the ability to access external C functions [3] from its built in KAL language by linking these C functions to the KAL function library. The C functions are written using the provided function library for knowledge management so that the functions already available in KAL can be made use of.

To accomplish a uniform handling of data, KAPPA uses the concept of atoms. An atom ID is a handle or reference to a location in memory where the actual data resides. The actual data can be in the form of int, char, long, or other C types. Functions such as KappaGetInt and KappaMakeInt are available to convert from the KAPPA atom types to the C types or vice-versa. All knowledge items including classes, instances, rules, goals and functions can also be referenced using atom IDs. Lists in KAPPA are represented much the same as atoms but have a unique list ID.

Each function defined in C is registered in the kappa.c using Register_Function. This function takes four arguments. The first argument is the name of the function which would be used in KAL, the second argument is the name of the equivalent C function which can be the same or different from the first argument. The third argument specifies whether arguments passed to the C function from the KAL function are evaluated before passing, and the last argument specifies the help category under which the new KAL function would finally be listed.

The registering of functions can be done in three different parts of the kappa.c file. If the statement is included in the WinMain() function, before the last return statement, the function is registered after KAPPA has first been loaded. To register the same function, whenever a new application is started by selecting 'New' from the 'File' menu, the statement has to be repeated in the InitNewApplication() function. If the function also has to be called when an application is opened using 'Open' from the 'File' menu, it has to be repeated in the InitOpenApplication() function. Thus, for most functions, the registering process has to be done thrice, once in each function. The modified kappa.c file for string and file enhancements can be seen in APPENDIX A.

All C functions are passed only one argument by KAL. This argument is a pointer to a data structure, called argument list, that contains the actual arguments. Functions are available to give the count of arguments in this list and to extract them from the list.

The return value of a C function has to be either an atom or a list ID. Functions exist to convert strings, integers and other C values into atom or list IDs. Values of true, false or error messages can also be returned to the calling KAL function.

Functions for knowledge handling, drawing, updating, printing images, customizing sessions, system calls and providing a rudimentary Dynamic Data Exchange (DDE) interface are also available. Once the functions are written and registered, the make file can be run to create the new enhanced KAPPA executable file.

CHAPTER IV

THE ENHANCEMENTS TO KAPPA

Existing String and File Functions

The functions in KAPPA can broadly be divided into eight categories. The knowledge functions manipulate classes, instances, slots, methods, rules, goals and user functions. The other functions are for handling logic, math, lists, strings, files, windows and for miscellaneous controls.

The string and file functions pertaining to text files are rather limited in KAPPA. The only string functions present in KAPPA 1.1X [2] are '#' for the concatenation of two strings, '#=' for checking the equality of two strings and FormatValue for formatting any string according to existing C conventions such as used in the C printf statement. There exist no string functions to parse strings, change their case, find their length or any other manipulations.

The file manipulation functions are mainly of two types, those manipulating text files and others which provide hooks into dBase and Lotus 1-2-3 for the manipulation of '.dbf', '.wks' or '.wk1' data files.

Text file functions exist to open files for reading and writing and to close them. The only function available to read the files is ReadWord, which can be used to read one word at a time. Each word is delimited by a space, tab or return character. Special

characters are read as stand-alone words. Quoted strings are read as a single word. If an optional length is added, the given length of characters is read in as one word. There is no easy way of reading one line at a time, unless each line is of fixed length. It is also not possible to directly read in fields delimited by a given delimiter.

Functions that perform write operations to text files include `WriteLine` to write any given line, `WriteQuoted` to write the line with quotation marks around it so that fields with embedded spaces may be read back as one word and other write functions to write knowledge base functions. The write function is easier to use, as it can be used along with the `FormatValue` function to write any type of formatted text.

Thus, for a knowledge base that exists as a normal ASCII text file, the functions required to manipulate and use this knowledge are limited. To enhance these limited functions, several other general purpose string and file functions written in C, using Windows functions in some cases, are interfaced with KAPPA to provide it added functionality in these areas.

The New Functionality

In view of the disadvantages of limited string and file functions, several new functions have been implemented, described in detail in APPENDIX B.

The string functions have been enhanced by adding fourteen general purpose object handling functions for handling the strings contained in slots of instances and classes. In most cases, these functions also manipulate and obtain information on classes and instances too. The functions added include `ObjectBreak` and `ObjectSegment` to parse strings. `ObjectLength` is used to find the length of any string or the number of instances

in a class. `ObjectPosition` gives the position of a sub-string within a larger string or position of an instance defined within a class. `ObjectToUpper`, `ObjectToLower` and `ObjectToProper` give KAPPA the ability to change the case of any string, class or instance. `ObjectToAnsi` can give the Windows ANSI equivalent number of the first character in the string while `ObjectToChar` gives the equivalent character for an ANSI value. The Windows ANSI set differs from the DOS ASCII set for the extended codes. `ObjectToRadix` can convert a number in a given base to any other base between 2 and 36. To sort lists of objects, the `ObjectSort` function can be used and the lists can also be indexed and searched using the `ObjectIndex` and `ObjectbSearch` functions. These index and binary search functions can be used with a primary and secondary key. Lastly, `ObjectInterpret` evaluates and returns the value of any valid KAL expression passed to it.

The file functions relating to text files have also been enhanced by adding eleven more functions. The file functions include `FileOpen` and `FileClose` to open a maximum of 20 different text files. Further, the ability to open a file such that the original read and write functions can also be used is also present. Files can also be opened to append and modify them. The `FilePosition` function can return the current position of the file pointer denoting the number of bytes already read, while `FileGoto` can set this pointer to a new value. The `FileDir` command can display or create a list of files for the user to choose from. Files can be deleted using `FileDelete` or renamed using `FileRename`. The `FileExec` function can be used to execute any DOS or Windows executable file with different windowing options for Windows executable files. Finally, `FilePrint` can be used to print any output file created. The printing ability gives KAPPA the ability to print to any printer selected under Windows, with a variety of features.

In most functions, the object:slot pair can also be written as 'object, slot'. A point to remember is that object:slot pairs are evaluated by KAL before being handed over to the C functions. Thus, the value in the object:slot pair is passed to the function, rather than the name of the object and slot, whereas no evaluation is done when they are separated by a comma. If an object:slot name is going to obtain a value from the function, it can be embedded in quotes as ' "object:slot" ' or passed separately as 'object, slot'. Further, object:slots evaluating to lists cannot be passed to C functions but can be passed separately by delimiting with a comma. In many cases, for string functions, if the string specified is an object such as a class, it is handled differently, giving the function a dual purpose.

Salient Features of Some String Functions

The ObjectSort function uses the heap-sort algorithm [14, 15]. This method of sorting was preferred over alternate forms of sorting as it gives the best overall run time. This sorting technique has an average and worst time order of complexity of $n \log(n)$ giving KAPPA the ability to sort lists quickly and efficiently, as list handling is slower than object handling in KAPPA.

If physical sorting is not necessary, the list can also be indexed. Indexing a list creates another list containing the positions of the original list, placed in a sorted order. This sorting of the key positions is done using a type of insertion sort [13, 15]. The new list is of numbers, making it easier and faster to use, especially when many lists need to be sorted on a single primary-key list. It also has the ability to sort the key positions using a secondary-key list.

A binary search algorithm [14] has also been implemented, in the ObjectbSearch function. It can do the binary search on a sorted list or the original list combined with the indexed list. This provides a fast search technique for lists in KAPPA, which was not possible before. Using a secondary key with a second list is also made possible when using indexed lists for a binary search.

Salient Features of Some File Functions

The FileOpen function can open up to twenty different files. This number has been chosen arbitrarily and can be changed to open more or less files by changing the MaxFiles value, defined in the enhanced C file. The function uses Windows capability to detect that a file to be opened in the read mode exists or not and if it does not exist, prompts the user to place the disk in drive A and continue or cancel, using a Windows dialog box. The function also has the ability to display a standard file selection dialog box, if the file specified contains wild-card characters. This has been incorporated by calling a Windows function written in C and exporting it in the kappa.def file. By declaring two external pointer variables called infile and outfile, used by KAPPA, the function links these file pointers to a file opened with numbers 0 or 1, so that the original read and write functions of KAPPA can be used.

The FileRead function reads a buffer-full of characters at a time to make it faster than reading them character by character. The buffer is then checked for fields and records, which are returned. A string matching function was chosen rather than character matching so that record and field delimiters need not be limited to a single character. Before the function exits, it sets the pointer back to the actual number of bytes read from

the buffer. If both, the record delimiter and the field delimiter, are NULL, this function reads a string of 200 characters at a time. The buffer limit was chosen to be 200 as this is the limit of any string in KAPPA. To read an entire list into KAPPA, as many records as possible are read into the buffer, until the buffer contains an incomplete record, in which case, more characters are read into the buffer and the process repeated. As most text file knowledge bases also contain comments for the user, the function has the ability to recognize these comment lines using the first character in a record and discarding the entire record if found to be a comment.

Disk read and write operations take up a lot of time. Windows is a multi-programming environment, but each application has to release control to Windows so that it may give control to other applications. KAPPA lacks this feature, making it impossible for the user to change or run another application while KAPPA is executing, except while waiting for user input. The new read and write functions have the capability to yield control to Windows while reading or writing from files. This makes it possible to run the read or write operation in the background, giving the user a chance to work on other applications in the meantime. It also has the ability to break out of the read or write operations, without first completing them entirely.

The FilePrint function uses a lot of the Windows functions and capabilities to print to the default Windows printer. Like the FileOpen function, it too can take a wild-card argument for a file name and display a dialog box from which a file can be selected to print. A function procedure, AbortProc, used by Windows to abort print attempts has also to be declared in kappa.def. Windows functions to send tabbed text out and print them in various styles are used in the FilePrint function. Multi-column output

is also possible. This is implemented by calculating the length of the largest line in the previous columns and starting another column only if space exists to accommodate the largest line after leaving a right margin. If any line in the last column exceeds the largest line calculated in the previous columns, this line may be truncated at the right margin. The number of multiple columns it creates depends upon the point size of characters used and the length of the longest line in the column.

Salient Features of Other Miscellaneous Functions

Two other small Windows functions have also been implemented. The RunModule function gives KAPPA the ability to start running a module directly, as soon as it is opened. It first opens and displays the session window if it is not already open. It then calls a function called Start. If this function does not exist, no other action is taken, but if it does, the function begins executing, thus making it possible to start an operation without manually invoking it. This works only when a file saved in the binary format is opened. A file saved in the ASCII text form is not affected. This function is called after registering other functions and is used only in the InitOpenApplication function of the kappa.c file. RunModule is not registered, hence it cannot be called from within KAPPA by the user; it is only used by the system.

The other function, RunBackground, gives KAPPA the ability to run non user-interactive applications in the background by yielding control to Windows so that it may run other applications. As explained before, KAPPA does not have the ability to yield control unless waiting for user input. This function makes it possible to yield control to Windows, so that other applications are given a chance to run. If no other

applications require attention, Windows yields control back to KAPPA. To use this function, it has to be included in the main processing loop of any KAPPA application, as this function polls for messages from other applications. Thus, it can be used with any loops, including loops using the original read and write file functions.

Other Miscellaneous Enhancements in KAPPA

Further enhancements were made in KAPPA by using Borland's Resource Toolkit [12] to modify directly some of KAPPA's existing resources and add some others.

A dialog box resource is added to display and select files for wild-card file arguments in the FileOpen and FilePrint functions. This is done by adding another FILE_SELECT resource to the other resources using the toolkit.

Dialog boxes for displaying PostMenu and PostMultipleSelection function choices in KAPPA are also modified. The PostMenu function of KAPPA has two different forms. When the number of selection items is small, the items are displayed in a Windows menu-like form, whereas when the number of items is too large to display on one screen, a Windows dialog box is called up to list the items. PostMultipleSelection is a dialog box similar to the PostMenu dialog box, but it provides the capacity to select multiple items. The width of these displays is limited, and they cannot be moved or resized.

These dialog boxes have been made wider so that longer strings can be displayed within them. Further, features such as sizing and moving, which were not possible before, have also been added. By adding a caption bar, the dialog boxes can be moved to

other parts of the screen so that areas of information covered by the dialog boxes can be seen by adjusting the size or moving them around.

The ability to sort the items displayed in the PostMenu dialog box has also been added. Given a list, the PostMenu function sorts this list and then displays it. The PostMultipleSelection dialog box now has extended keyboard and mouse capabilities provided by Windows for multiple selection. Multiple items could not be selected by dragging the mouse or using it with the shift and control keys.

All the above adjustments can be made easily, using Borland's Resource Toolkit, which is menu driven. The corrections can be made to the resource file and then linked together to form the KAPPA executable file or the changes can be made after creating the executable file by directly modifying this executable file. Modifying the resource file is more convenient as it is smaller, faster to modify and can be included while compiling, whereas the executable file is much larger, taking longer to modify as it requires modification each time the executable is changed.

CHAPTER V

COMPARISON AND ANALYSIS OF ENHANCEMENTS IN KAPPA

Comparison of Old and New Features

The KAPPA PC string functions are very limited. They do not have the ability to break or parse strings, change case, sort, index or do a binary search on lists. For example, if the telephone number of a company exists in the knowledge-base, it is quite impossible to separate the area code from the rest of the number. Sometimes, it is necessary to obtain the initials of a person whose name is stored as a string. The new functions `ObjectLength`, `ObjectPosition`, `ObjectSegment` or `ObjectBreak` can be used singly or in combination with the other functions to produce the desired results.

Similarly, occasions can arise where knowledge stored with strings in all capital letters needs to be printed in lower-case or with the first letter capitalized. The functions `ObjectToUpper`, `ObjectToLower` or `ObjectToProper` can be used for this purpose.

KAPPA PC does not have the ability to sort or index an unsorted list so that it can be printed or displayed in sorted order. The `ObjectSort` or `ObjectIndex` functions can be used for this purpose. There is also no function to do a binary search on a sorted list. There are, however, sequential search functions such as `Member?` and `SelectList`, [2] that search the list sequentially until the given pattern is matched. A binary search is usually

quicker for sorted lists. User-defined functions can be written in KAL to sort, index or do a binary search. A disadvantage of doing so is that the time and effort is wasted in writing similar functions whenever needed. Further, user-defined functions are interpreted, and hence are slower than functions encoded in the binary form. The enhanced functions produce a higher level of abstraction for developing applications requiring sorting.

Other string functions to find ANSI equivalents of characters and characters for ANSI values are also available in the enhanced version. The ObjectToRadix function changes the value of any given number, in any base between 2 and 36, to any other base in the same range. These functions cannot be user-defined in KAL and are useful when using or displaying technical data.

In enhancing the file functions, new, open, close, read and write functions have been written to augment the ones already present. A maximum of one read file and one write file could be opened using KAL functions OpenReadFile and OpenWriteFile [2]. It is now possible to open a maximum of 20 files using the generalized FileOpen function in the enhanced version. These can be opened in a combination of read, write, modify or append modes. The files could not be opened for modifying or appending originally. To help modify files, the FilePosition and FileGoto functions give or change the current position of the file pointer.

If a file is opened in the read mode with the file number 0 or in the write mode with the file number 1, it is similar to opening the files using OpenReadFile and OpenWriteFile respectively, except that they are opened for binary reading or writing rather than text. In this binary mode, if the Write functions are used, the end of line is

written by the line feed character '\n' without the carriage return '\r', which then has to be placed using the FormatValue function. For example,

```
Writeline(FormatValue("Write this line.\r\n"));
```

writes the line correctly.

As the original read and write KAL functions use buffers, it is not advisable to mix these statements with the new read and write functions, though the read statements are interchangeable if used with care. The write statements give unpredictable results, requiring the use of the CloseWriteFile function to flush the buffers.

The original ReadWord function in KAL is capable of reading just one word at a time. Each word is delimited by a space, tab, return or other special character. Special characters are read as words by themselves. Lines containing words, spaces or other special characters, enclosed within double quotes are also read as a single word. An optional numeric value makes ReadWord read a fixed number of characters as one word. There is no provision to read an entire line at a time or to divide the text file into records and fields. The new FileRead function provides these capabilities.

The new FileRead function can read and discard comment records which are specially delimited records, placed in the text knowledge-base for readability. The records and fields can be more than one character long. Multiple field or record delimiters and their last characters repeated in tandem are ignored. For example if the field delimiter is "abc", the entire field delimiter "abcabcabcccc" will be skipped before reading the next word. Thus a field delimiter defined as three spaces will find words separated by three or more spaces.

A user-defined function can be written in KAL to produce similar results. This function has to read the whole file character by character by making each word of one character length. Each character then has to be checked and compared with the field and record delimiters and concatenated together if not a delimiter. If the field or record delimiters contain more than one character, it can get very complicated to find these delimiters without other string handling features being available. Such an analysis, done on some types of read functions, is given later.

File functions have been added to delete, rename, obtain, execute and print file names. For example, if more than one input file has been created and the user has to be given a choice to read one of these files, the FileDir function can be used to display a standard dialog box for displaying wild-card characters and after selection, the file can be manipulated. Further, if a multiple selection is needed, all the files matching the criteria can be placed into a list and later manipulated. This list can also be used for multiple file deleting or renaming. User-defined functions in KAL cannot be written to emulate these functions.

The existing Execute function has certain limitations. It can run any DOS or Windows executable file, but when running a Windows executable file from within a function, a user input screen that follows the Execute function call gets linked to the new window such that the user is unable to make the new window active. To avoid this, the user can hide and recall the session window after the Execute function is called, thus making the session window active and linking the user input dialog box to the session. But in such a case, the newly executed window is hidden behind the current session

window, and the user has to once again call it up using the ALT-TAB or similar Window shortcut key.

The new FileExec function is written so that the new window is displayed and the user can move to that window directly by clicking his mouse on the window. An added facility to minimize, maximize or place the new window in the background has also been added.

An output created in KAPPA and saved as a file is difficult to print. KAPPA has no built in function to print a text file on a Windows defined printer. One way to print a text file is to call the Execute function to run another executable file which then takes the output file as an argument name and prints it out. One could also change applications within Windows to one which can print the text file out. The only other way is to write to a file opened with a DOS device name such as PRN, LPT1 or COM1.

The FilePrint function which has been added to KAPPA gives KAPPA the ability to print a text file to the default Windows printer. The 'Control Panel' application of Windows can be used to define or change the Windows default printer. Optionally, the name of a Windows printer can also be given as the last parameter. If the name contains an '*' or '?', a list of a maximum of ten printers, defined in Windows, is displayed to choose from. However, the selected printer has to be active. Provisions exist to change the font size, font name and select options such as printing in bold, italic, underline, with fixed or variable fonts, without margins, setting the right margin to a maximum of n characters per line (counts a tab as a single character rather than eight blanks) with longer lines wrapped to the next line, with a header displaying the name of the file and current date, with page numbers, line numbers and with multiple columns per page. The options

are declared using a special character for each option, which can be concatenated together to form a word, thus making the function compact.

Consequences of the New Functionality

The new functions added to KAPPA have been made as versatile as possible. These functions can handle both single or multi-valued slots (lists) using the same function name and format. The format has been made adaptable to both methods of denoting values in slots, using 'object:slot' or 'object, slot'. In many cases, subtle differences exist between these two formats so that while the former makes the function return a result, the latter acts more like a procedure by storing the result back into one of the arguments. Default values with less or no parameters are also accepted in most functions to make them shorter and more space efficient.

Due to the addition of these functions and their various features, the overall size of KAPPA has increased. Whereas, the original size of the KAPPA 1.1X file was about 565 K bytes, the enhanced version is 686 K bytes, a difference of 121 K bytes. This increase has taken the limit of the KAPPA executable above that which can be loaded into conventional memory. Thus, a minor drawback with the enhanced version, is its inability to be loaded, if Windows is running in the real mode, as Windows does not use memory above 640 K bytes in this mode. The enhanced version of KAPPA requires extended memory. There is no significant increase in loading time due to this increase in size.

Comparison and Analysis of User Defined Functions With and Without the New Functionality

Some of these enhanced functions provide an easier way of writing certain applications, which are tedious but possible to do, using the old KAL functions. Other functions would be all but impossible to write using the old KAL functions without making external calls to other executable files. A few others would not be possible in either case. All of them, certainly could be created using the C interface. Each of these methods is described below with examples, and an analysis of the advantages and disadvantages in each case is made.

Working with Old KAL Functions

User defined functions where the newer functions could be used to advantage but are not be absolutely necessary, mainly consist of the text file handling functions. The new open and close functions are not essential if only one read or write file is opened at any given time, and the file is neither modified nor appended to. The new write function provides some marginal benefits when files with fixed field and record delimiters need writing to. The old write function, along with the FormatValue function can be just as advantageous.

The old read function, on the other hand, is pretty weak when reading records with fields within them, specially when the field contains spaces or special characters within them. A function can be written to do this, but would either be extremely slow due to a lot of extra processing, or could be made faster but not as accurate.

Two such examples of data files, taken from an expert system project for configuring industrial filtration systems, are shown in Figure 6.

<pre>***** * Example1.TXT * SELECTION LIST FOR * CUSTOMERS AND * DISTRIBUTORS * * Address Code ***** CLASS CUSTSEL ABC Company, abc st., NY A1 AXE Company, axe st., TX A2 XYZ Company, xyz st., FL X1 ZZZ Company, zzz st., AL Z1</pre>	<pre>***** * Example2.TXT **** SALES ENGINEERS **** CLASS SALES SALES_AV Bob Smith BS Tom Little TL Suzy Thomas ST SALES_IND Greg Ramsey GR Bill Fairbanks BF Roy Jones RJ **** REGIONAL MANAGERS **** CLASS REGMAN REGMAN_SE Joe White, Regional Manager XYZ Company Columbus, Ohio 45002 (513) 561-1111</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 6. Example Data Files

The user defined functions to read these files, using the original and enhanced versions of the read functions are shown in Figure 7 and Figure 8. Example1 requires the data to be read into a list. The unenhanced version to do this, reads and stores a fixed number of characters into the list as the address. This method is fast, but less accurate in that short strings have unnecessary blanks at the end of each string. The unenhanced

version in Example2, to read values into slots, is more accurate in reading single spaces within strings and finding the end of each line, but this increases its size and speed.

<pre> /***** ** Load_Example1 without Enhancements *****/ MakeFunction(Load_Example1, [], { ResetClock(); CatchError(CloseReadFile()); CatchError(OpenReadFile(Example1.TXT), PostMessage("Example1.TXT not found.")); While (Not(Null?(Global:Tmp0 = ReadWord()))) { If (Global:Tmp0 #= *) Then ReadWord(65) Else{ If (Global:Tmp0 #= CLASS) Then{ Global:Tmp = ReadWord(); CatchError(MakeClass(Global:Tmp, Customers)); ReadWord(2); }; MakeSlot(Global:Tmp:Address); SetSlotOption(Global:Tmp:Address, MULTIPLE); ClearList(Global:Tmp:Address); MakeSlot(Global:Tmp:Code); SetSlotOption(Global:Tmp:Code, MULTIPLE); ClearList(Global:Tmp:Code); Global:Tmp0 = ReadWord(33); While (Not(Null?(Global:Tmp0) Or Number?(Global:Tmp0 # 0)) { AppendToList(Global:Tmp:Address, Global:Tmp0); AppendToList(Global:Tmp:Code, ReadWord()); ReadWord(1); Global:Tmp0 = ReadWord(33); }; }; CloseReadFile(); PostMessage(GetClock(), " secs."); }); </pre>	<pre> /***** ** Load_Example1 with Enhancements *****/ MakeFunction(Load_Example1, [], { ResetClock(); FileClose(0); If(FileOpen(0,Example1.TXT,READ)#=FALSE) Then PostError("Example1.TXT not found"); While (Not(FileRead(0, *, " ", "\n") #= CLASS)) {}; FileRead(0, *, " ", "\n", Global, Tmp); CatchError(MakeClass(Global:Tmp,Customers)); MakeSlot(Global:Tmp:Address); SetSlotOption(Global:Tmp:Address, MULTIPLE); ClearList(Global:Tmp:Address); MakeSlot(Global:Tmp:Code); SetSlotOption(Global:Tmp:Code, MULTIPLE); ClearList(Global:Tmp:Code); While (Not(FileRead(0, *, " ", "\n", Global:Tmp, Address, Global:Tmp, Code) #= FALSE)) {}; FileClose(0); PostMessage(GetClock(), " secs."); }); </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 7. Reading Example1.TXT

<pre> /***** ** Load_Example2 without enhancements *****/ MakeFunction(Load_Example2, [], { ResetClock(); CatchError(CloseReadFile()); CatchError(OpenReadFile(Example2.TXT), PostMessage("Example2.TXT not found.")); While (Not(Null?(Global:Tmp0 = ReadWord()))) { If (Global:Tmp0 #= *) Then ReadWord(80) Else{ If (Global:Tmp0 #= CLASS) Then{ Global:Tmp = ReadWord(); CatchError(MakeClass(Global:Tmp, Customers)); Global:Tmp0 = ReadWord(); }; CatchError(MakeInstance(Global:Tmp0, Global:Tmp)); MakeSlot(Global:Tmp0:Name); SetSlotOption(Global:Tmp0:Name, MULTIPLE); ClearList(Global:Tmp0:Name); If Global:Tmp #= SALES Then{ MakeSlot(Global:Tmp0:Initial); SetSlotOption(Global:Tmp0:Initial, MULTIPLE); ClearList(Global:Tmp0:Initial); }; Global:Tmp2 = ReadWord(); ReadWord(1); For y [1 Global:Tmp2] { If Global:Tmp #= SALES Then{ Global:Tmp4=""; While (Not((Global:Tmp3 = ReadWord(1)) #= FormatValue(" "))) { Global:Tmp4 = Global:Tmp4 # Global:Tmp3; While (Not((Global:Tmp3 = ReadWord(1)) #= FormatValue(" "))) Global:Tmp4 = Global:Tmp4 # Global:Tmp3; }; AppendToList(Global:Tmp0:Name, Global:Tmp4); While ((Global:Tmp3 = ReadWord(1)) # = FormatValue(" ")) { }; AppendToList(Global:Tmp0:Initial, Global:Tmp3 # ReadWord()); ReadWord(1); } Else{ Global:Tmp4=""; </pre>	<pre> While (Not((Global:Tmp3 = ReadWord(1)) #= FormatValue("\n"))) Global:Tmp4 = Global:Tmp4 # Global:Tmp3; AppendToList(Global:Tmp0:Name, Global:Tmp4); }; }; }; }; CloseReadFile(); PostMessage(GetClock(), " secs."); }); /***** ** Load_Example2 with enhancements *****/ MakeFunction(Load_Example2, [], { ResetClock(); FileClose(0); If(FileOpen(0,Example2.TXT,READ)#=FALSE) Then PostError("Example2.TXT not found"); While (Not(FileRead(0, *, " ", "\n", Global, Tmp1, Global, Tmp2) #= FALSE)) { If (Global:Tmp1 #= CLASS) Then{ Global:Tmp = Global:Tmp2; CatchError(MakeClass(Global:Tmp, Customers)); } Else If (ObjectBreak(Global:Tmp1, _) #= Global:Tmp) Then{ Global:Tmp0 = Global:Tmp1; CatchError(MakeInstance(Global:Tmp0, Global:Tmp)); MakeSlot(Global:Tmp0:Name); SetSlotOption(Global:Tmp0:Name, MULTIPLE); ClearList(Global:Tmp0:Name); If (Global:Tmp #= SALES) Then{ MakeSlot(Global:Tmp0:Initial); SetSlotOption(Global:Tmp0:Initial, MULTIPLE); ClearList(Global:Tmp0:Initial); }; } Else{ AppendToList(Global:Tmp0:Name, Global:Tmp1); If (Global:Tmp #= SALES) Then AppendToList(Global:Tmp0:Initial, Global:Tmp2); }; }; FileClose(0); PostMessage(GetClock(), " secs."); }); </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 8. Reading Example2.TXT

In the above examples, the word following the identifier CLASS is the class name which is read, created and the name saved in Global:Tmp for later use as the class name. In Example1, a fixed field of characters is read and appended to the multi-valued slot, Address, declared under the class which has just been read. The code is read as a word and appended to another list. Reading a fixed length of characters, such as for the address, requires a fixed field length to be maintained and unnecessarily adds spaces at the ends of shorter lines. These extra spaces also cause needless blanks to be displayed in PostInputForm dialog boxes, where the display of slot values is limited to the last few characters.

The class name for the enhanced version is found by essentially reading word by word from the file until this keyword is encountered upon which the next word read gives the name of the class, which is again stored under Global:Tmp. The entire list reading operation, can be performed by a single FileRead statement within a while loop, as shown in the enhanced version. The field delimiter is considered to be two or more spaces and the record delimiter is the return key.

In Example2, the unenhanced version tries not to read fixed length fields with unnecessary spaces. To do this, it has to read the file one character at a time and match it with a space. When a space is found, it then has to check if the next character is also a space, in which case, it has found the field, otherwise, the blank has to be considered as part of the old field. This process requires a number of 'While' loops, and slow character by character reading of the file. It is also very difficult to find the end of a line, when each line is of a different length. This is achieved by using the FormatValue function to trap the return character and check it.

The enhanced version, on the other hand, is comparatively shorter and easier to write and uses just one FileRead function call, compared to 12 separate ReadWord function calls in the unenhanced version. Hence, the time taken to read a file using the enhanced functions is far less than that without the new functions.

The system configuration used for all the above comparisons was a 33 MHz, 80386 PC with 8 MB of RAM and 128 K cache, running DOS 5.0 and Microsoft Windows 3.0. A graphical representation of the comparative sizes and speed of the functions is given in Figure 9 and Figure 10.

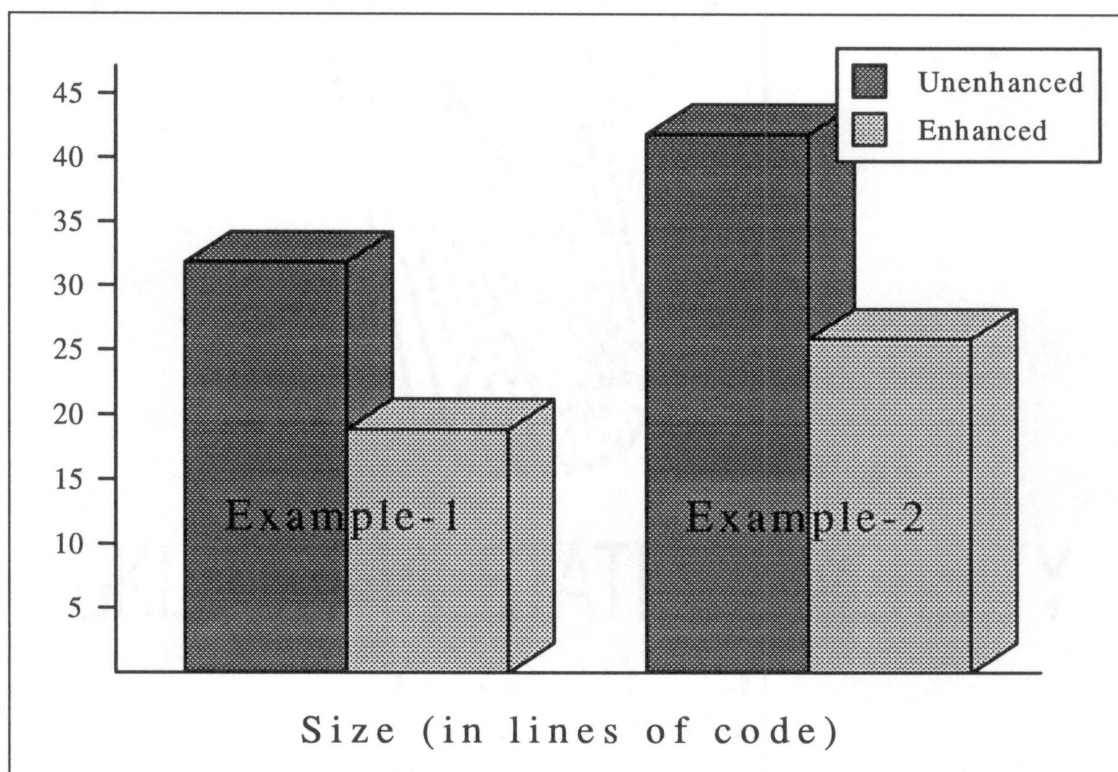


Figure 9. Comparison of Code Sizes

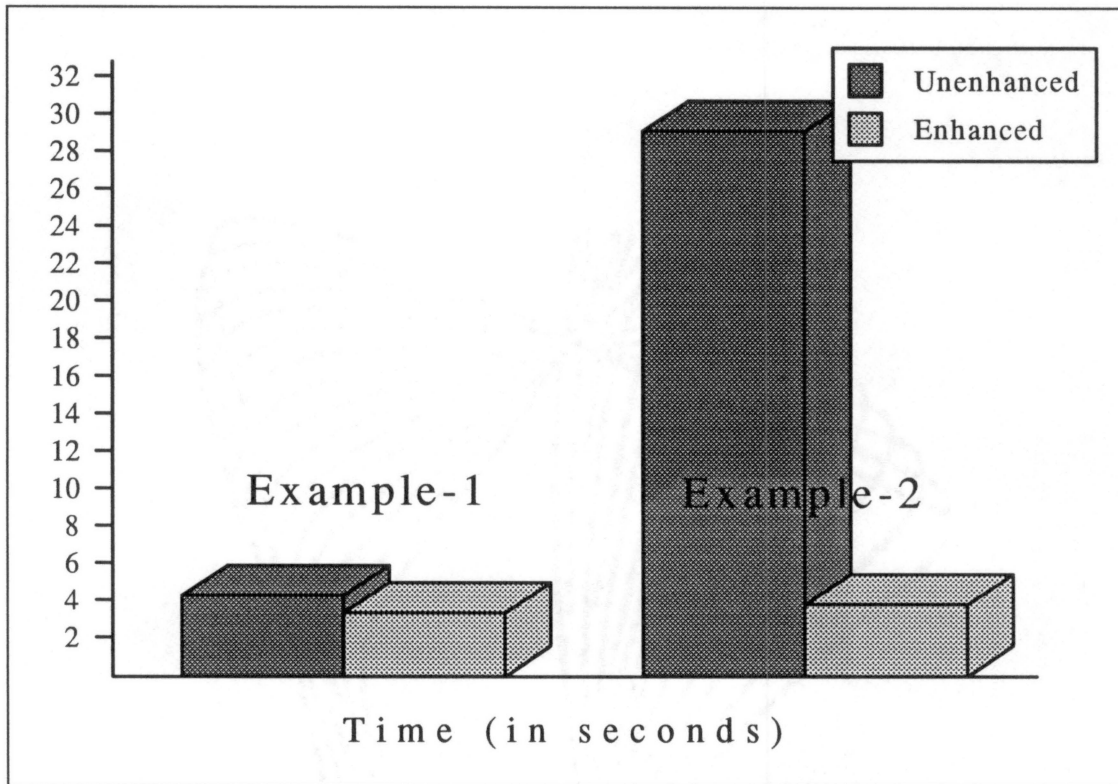


Figure 10. Comparison of Execution Times

Calling External Executable Files

When there is no KAL workaround to produce similar results of functions using the enhancements, which is the case for all the string handling functions and some special file handling functions, an outside executable file can be called to do this. The executable file can either be of DOS or Windows and can be called using the Execute function present in KAL. This executable file has to be already existing software or can be created using other language compilers or assemblers.

The main drawback of using this method is that it is slower as an external function call has to be made. Another assembler or compiler and linker is required and

has to be available to create this executable file. Using the in-built functions can be faster. Passing of results back to the calling program is also a major problem.

It is not easy to return the results of the calling program back to KAPPA.

Arguments can be specified to pass values to the executable file, and if no return values are required, no further action need be taken. For example, to delete or rename a file from KAPPA, the execute command could be given as:

```
Execute("command.com /c", del, "test1.bak");           OR
```

```
Execute("command.com /c", ren, "test1.bak", "test2.bkp");
```

The execute function always returns a TRUE whenever the given executable file is found. Thus, even if there were no "*.bak" files, the function would still return TRUE. There is no way of knowing the outcome of the command given. A command such as the one given above takes about 0.65 secs, whereas a direct call using the enhanced functions (FileDelete or FileRename) takes only 0.05 secs relatively, with the same configuration of the system as stated above.

In cases where the results from the executable file are required, a ready made executable program is not sufficient. A special function has to be written, compiled and linked. This function has to send the output, required by KAPPA, to an intermediary text file. This file can then be read in KAPPA, after returning from the Execute function call, to obtain the required results. This entails a much further loss of time and effort. A simple example of such a C file, to convert a given string to uppercase, and the KAL function required to call it and obtain the results, are given in Figure 11. The C file has to be compiled and linked to form an executable before it can be called in the KAL routine.

<pre> /***** C file to convert the first argument passed to it into uppercase and return this value via the data file tmp. *****/ #include <stdio.h> main(argc,argv) int argc; char **argv; { char st[200]; FILE *fp; fp = fopen("tmp", "w"); if (argc <= 1) fprintf(fp, "FALSE\n"); else fprintf(fp, "%s\n",strupr(argv[1])); fclose(fp); return(0); } </pre>	<pre> /***** KAL version to call and make use of the C function to convert a string to uppercase. *****/ MakeFunction(Test, [], { PostInputForm("Enter a lowercase string", Global:Tmp, "String:"); Execute(TEST.EXE, Global:Tmp); PostMessage("String in lowercase is ", Global:Tmp); OpenReadFile(tmp); Global:Tmp = ReadWord(); CloseReadFile(); PostMessage("String in uppercase is ", Global:Tmp); }); </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 11. C and KAL Functions to Convert a String to Uppercase

Another major disadvantage in using the Execute function is that due to the Windows multi-tasking nature, the statements immediately following the Execute function are executed before the Execute function itself is completed. If a dialog box such as a PostMessage or PostMenu is displayed immediately following the Execute function, it then gets the opportunity to reach completion.

In the above example, if the PostMessage function was not called immediately following the Execute function, the value read from the text file would be one which was existing prior to the Execute function or if it was the first time, the OpenReadFile function would give a 'file not found' error. The Execute function sometimes crashes the system after returning from the function call. The enhanced function FileExec does not

crash and a small loop of calls to the RunBackground function, immediately following the FileExec or Execute function, can allow the external function call to be completed before processing the output text file. Of course, the entire above example runs much faster by just calling the enhanced function, ObjectToUpper, as:

```
PostMessage("The string in uppercase is ", ObjectToUpper(Global:Tmp));
```

mainly because the function has been incorporated right into the KAPPA executable file, thus not requiring external calls.

Some enhanced functions such as the FilePrint, FileDir and FileOpen functions make use of the Windows dialog boxes. For these functions, if an external executable file is to be created, it requires the Software Development Kit for Microsoft Windows, [5, 6, 7, 8] or another compiler such as Borland C++, [9, 10, 11, 12] which has the capability of creating Window executable files. This requires extra effort on the part of KAPPA application developer to create and use these general purpose functions.

Interfacing with C into KAPPA

The last method to make enhancements is by interfacing with C to create a KAPPA executable file. This is the method which has been used to make all the enhancements in this thesis, as shown in the chapters above. If a function requires direct access to KAPPA dialog boxes, functions or other KAPPA related definitions, which are not present in KAL, this method provides the only means of defining them in KAPPA.

By already incorporating general purpose functions such as the file and string functions, right into the KAPPA executable file, the execution of these commonly used

functions becomes faster and more convenient. Once placed into the KAPPA executable file, the functions can be used very easily, just as any other KAL function.

The main disadvantage in this method is that the C interfaced code is incorporated into KAPPA and this increases the size of the executable. If the functions incorporated are not general purpose or though important are used infrequently, a lot of the code just occupies memory, to no useful purpose. This extra code may also slow down the system a little and may take up much of the heap, stack space and resources in Windows which could be required with larger applications. Thus the functions added to KAPPA should be as general purpose as possible so that they can be used in many different applications. If functions for certain rarely used functions need to be incorporated into an executable file, it can sometimes be more convenient to have different versions of the KAPPA executable available for the different needs.

Comparison with Newer Versions of KAPPA

String Functions

Some basic string handling features have been added to versions of KAPPA after 1.1X. The EvaluateKAL function of KAPPA 1.2 is similar to the enhanced KAPPA 1.1X ObjectInterpret function. They both evaluate any valid KAL expression to return its result. EvaluateKAL has better syntax error messages but an expression cannot be split into its components and given as multiple arguments, which is possible in the ObjectInterpret function.

The FindSubString function, in KAPPA 1.2 or above, is similar to the ObjectPosition function in the enhanced version of KAPPA 1.1X. These functions are to find the position of one substring within a larger string. The FindSubString requires three arguments whereas the third argument is optional in ObjectPosition. The third argument in the FindSubString function is the nth occurrence of the substring to search for, whereas the third argument in the ObjectPosition function is the position in the string from which to begin a search. One advantage of the FindSubString function is its ability to search backwards for the nth occurrence by giving a negative number. The ObjectPosition, on the other hand, can also handle lists to give the occurrence of an element in a list or calculate the position of an instance within a class.

The StringLength function of KAPPA 1.2 is similar to the ObjectLength function in the enhanced version. These functions return the length of any string given as an argument. The ObjectLength function can, in addition, give the lengths of lists and the number of instances in a class.

The SubString function of KAPPA 1.2 is similar to the enhanced ObjectSegment function. The SubString function takes three arguments while the third argument is optional in ObjectSegment. The third argument in SubString is the ending position of the string to be extracted, whereas the third argument of ObjectSegment is the length of the substring to be extracted. Further, if the third argument is omitted in ObjectSegment, the second argument becomes the length of the substring to be extracted from the starting position. The ObjectSegment function, unlike the SubString function, can also take lists or classes as arguments to extract sub-lists or a sub-group of instances within a class.

The TextCase function of KAPPA 1.2 is similar to the combination of the ObjectToLower and ObjectToUpper enhanced functions. The conversion to uppercase or lowercase can be carried out by the TextCase function, by stipulating the keyword, UPPER or LOWER. The TextCase function does not have the ability to change the case of an entire list, nor is it able to convert the case of classes and instances, unlike the enhanced functions.

File Functions

Some new functions have been added to the file functions relating to text files and other external files such as database, spreadsheet files and for Dynamic Data Exchange in KAPPA 1.2 and above. There are only two new functions that have been added for text file handling.

The ReadLine function of KAPPA 1.2 has been added to augment the functionality of the ReadWord function. The ReadLine function can return an entire line or a fixed number of characters in a line. The functions still cannot read separate fields or records delimited by other characters. A field would have to be given within double quotes, if it contained spaces or other special characters. The enhanced FileRead function is able to read fields and records delimited by one or more characters.

The only other file function added in KAPPA 1.2, relating to text files is the SelectFile function, which can display a dialog box for the user to choose files from. The complete path of the file chosen is returned by this function. The enhanced FileDir function is similar, but does not return the entire path. However, it does set the current path to one where the selected file exists. It is also able to return an entire list of files in

a directory and place them into a multi-valued slot, which is not possible using the SelectFile function.

Other Enhancements

The KAPPA 1.2 WaitForInput function is similar to the RunBackground enhanced function. WaitForInput allows the KAPPA program to yield to other applications, to process their messages, by handing control to Windows. This allows the other screens in KAPPA to also be updated. The enhanced RunBackground function does the same thing.

The enhancements made in C can also be interfaced into the newer version of KAPPA so that added functionality is given to the newer versions and they can remain compatible with applications using the enhanced functions in KAPPA 1.1X. The newer libraries and resources need to be combined with the enhanced C functions by compiling and linking them to form a new enhanced version of KAPPA 1.2 or higher.

CHAPTER VI

SUMMARY AND CONCLUSIONS

KAPPA provides an interactive graphical user interface which can be used to advantage in building applications and PC-based expert systems. It is a hybrid tool which runs under Microsoft Windows and provides the user with object oriented tools along with traditional rule based reasoning.

The file and string handling capabilities of KAPPA 1.1X were found lacking and have been enhanced in this research. Some of the added functionality is present in the newer versions of KAPPA but these versions still lack features such as being able to handle lists and objects as strings or manipulating multiple files. The enhanced functions give KAPPA a far more advanced capability when working with strings and text files.

Fourteen general purpose functions were added to manipulate character strings present in single-valued or multi-valued slots within KAPPA objects. These include functions such as ObjectBreak and ObjectSegment to parse strings, ObjectLength and ObjectPosition to find lengths and positions of strings, functions to change the case of strings, change numbers to other bases and get ASCII character codes. The ObjectSort, ObjectIndex and ObjectbSearch functions sort, index and search through multi-valued slots, to give additional list processing capabilities.

Eleven other general purpose functions to manipulate ASCII text files used as knowledge bases or for output purposes were also added. Functions to open, close, read and write to multiple files were incorporated. The enhanced read and write functions were given the capability to read and write files containing record and field delimiters. The read function also has the capability to read and discard comment lines or records present within text files and also yields to other Windows applications, so that the reading process may be performed in the background. Functions were also provided for random access to files by giving them the ability to find and set the position of the file pointer. Other functions to delete, rename, print and run files were also incorporated.

In addition, a function was also written to be able to run KAPPA in the background, while running other Windows applications in the foreground, by making KAPPA yield to Windows. This function can also force an outside executable file to run to completion, before KAPPA progresses, allowing KAPPA applications to use the new data created by the executable.

All the enhancements were made by interfacing C with KAPPA. The enhanced functions were made integrating C and Windows functions and then compiling and linking them with the KAPPA libraries and resources to form a new executable. A comparison and analysis were performed on these enhancements to show the advantages of adding these functions to KAPPA.

KAPPA provides easy access to all its built-in functions. These functions are provided in a file that can be included with C source code for new KAPPA functions. This allows the use of a rich mixture of C, Windows and KAPPA functions to write user-defined functions which can be incorporated into the KAPPA executable for ease

and speed. These user-defined functions can be registered and then called like normal KAPPA functions from within KAPPA.

The addition of new functions increases the size of the KAPPA executable. The enhanced version can thus no longer be used in Windows running in the real mode which is limited to 640 KB RAM. The analysis, through examples and graphs, shows that some of the enhanced functions can be written with existing KAPPA functions but would be tedious, slow and require lengthy programming effort. Some of the other enhanced functionality can be emulated by making external calls to outside executables from within KAPPA with similar handicaps, while some of the functionality cannot be achieved without following the procedures adopted in this research..

Finally, a comparison has also been made with the newer versions of KAPPA, which have incorporated additional functions that are similar to the enhanced functions of KAPPA 1.1X. The comparison shows that on the whole, the enhanced functions provide added capabilities and features to those available in newer versions. Some of the enhanced features do not have any equivalents in the newer versions. The possibility of adding these enhanced functions to the newer versions of KAPPA and increasing the overall functionality of these newer versions, has also been discussed and shown to be advantageous.

With all of the enhancements provided from this research, KAPPA still lacks certain features which could be added using the approach applied in this work.

KAPPA cannot deal with uncertainty. There are no functions for fuzzy logic, calculating confidence factors or probabilities. Some of these could be implemented using the C interface.

Functions to enhance further its external interface with file handling functions for other data base and spreadsheet files are also limited. General, basic DDE (Dynamic Data Exchange) protocol functions are also lacking in this version of KAPPA, though some limited functions are present in the newer version. These can be enhanced further by making use of the C interface using Windows provided functions.

The PostInputForm function which can display and change the values of a maximum of ten different slots on the same screen is very useful. The windows displaying the values in these slots are rather small and lack the capability to expand these edit areas so that lengthy text can be displayed and modified. Further, only one PostMenu or other type of dialog box to display lists, can be shown at one time. It is not possible for KAPPA to show two or more independent or linked information columns at the same time, for the user to choose from. Using the C interface and the Windows functions, it would be possible to write input function menus based on the above, with far more features and wider edit windows.

For a PC-based expert system shell, KAPPA delivers a lot of power. The addition and implementation of the enhancements makes KAPPA an even stronger and more user friendly tool to work with. Interfacing the enhanced functions with newer versions of KAPPA can give it added functionality and keep it compatible with the older, enhanced versions.

BIBLIOGRAPHY

1. "KAPPA User's Guide", KAPPA Version 1.0, IntelliCorp Inc., May 1990.
2. "KAPPA Reference Manual", KAPPA Version 1.0, IntelliCorp Inc., May 1990.
3. "KAPPA C Interface Manual", KAPPA Version 1.1, IntelliCorp Inc., July 1990.
4. "Microsoft Windows 3.0, Graphical Environment, User's Guide", Microsoft Corporation, 1985-1990.
5. "Microsoft Windows Software Development Kit, Reference--Volume 1", Version 3.0, Microsoft Corporation, 1990.
6. "Microsoft Windows Software Development Kit, Reference--Volume 2", Version 3.0, Microsoft Corporation, 1990.
7. "Microsoft Windows Software Development Kit, Guide to Programming", Version 3.0, Microsoft Corporation, 1990.
8. "Microsoft Windows Software Development Kit, Tools", Version 3.0, Microsoft Corporation, 1990.
9. "Borland C++ Users Guide", Version 2.0, Borland International, 1991.
10. "Borland C++ Programmers Guide", Version 2.0, Borland International, 1991.
11. "Borland C++ Library Reference", Version 2.0, Borland International, 1991.
12. "Borland C++ Whitewater Resource Toolkit", Version 2.0, Borland Intl., 1991.
13. Herbert Schildt, "C The Complete Reference", Osborne McGraw-Hill Inc., 1987.
14. J. P. Tremblay, and P G. Sorenson, "An Introduction to Data Structures with Applications", McGraw-Hill Inc., 1984.
15. D. E. Knuth, "Sorting and Searching. The Art of Computer Programming. Vol 3", Addison-Wesley Publishing Co. Inc., 1973

16. Thomas Helton, "Object-Oriented Expert-System Tool, Kappa-PC 1.1", Software Review, AI Expert, March 1991.
17. T.J. Lydiard - University of Edinburgh, "Kappa-PC", Product Reviews, IEEE Expert, October 1990, 71-77.
18. Won Kim and Frederick H. Lochovsky, "Object-Oriented Concepts, Databases, and Applications", ACM Press, Addison-Wesley, September 1989.
19. O. J. Dahl and K. Nygaard, "SIMULA - a goal-based simulation language", Communications of the ACM, Vol 9, 1966, 671-678.
20. G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard, "Simula Begin", Auerbach Press, Philadelphia, 1973.
21. A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983.
22. Wilf R. LaLonde and John R. Pugh, "Inside Smalltalk", Vol I, Prentice Hall, 1991.
23. Wilf R. LaLonde and John R. Pugh, "Inside Smalltalk", Vol II, Prentice Hall, 1991.
24. Bruce G. Buchanan and Edward H. Shortliffe, "Rule-Based Expert Systems. The MYCIN Experiments of the Stanford Heuristic Programming Project", Addison-Wesley, 1984.
25. Ken Pedersen, "Expert systems programming: practical techniques for rule-based systems", Wiley, New York, 1989.
26. Amar Gupta and Bandreddi E. Prasad (editors), "Microcomputer-Based Expert Systems", IEEE Press, 1988.
27. Ralf Alberico and Mary Micco, "Expert Systems for Reference and Information Retrieval", Meckler Corporation, 1990.
28. Paul Siegel, "What Expert Systems Can Do", Training & Development Journal, Vol 43, Iss 9, September 1989, 70-73
29. Thomas C. Bartee (editor), "Expert Systems and Artificial Intelligence. Applications and Management", Howard W. Sams & Company, 1988.
30. Susan Lindsay, "Practical Applications of Expert Systems", QED Information Sciences, Inc., 1988.

31. David Stamps, "Taking an Objective Look", *Datamation*, Vol 35, Iss 10, May 1989, 45-48.
32. David Horn, "Expert Systems Emerge from Their Shells", *Mechanical Engineering*, Vol 111, Iss 4, April 1989, 64-67.
33. Martin Ramsey, "Gaining Proficiency in Expert Systems", *Mechanical Engineering*, Vol 111, Iss 4, April 1989, 73-78.
34. Zack Urlocker, "Object-Oriented Programming for Windows", *Byte*, Vol 15, Iss 5, May 1990, 287-294.
35. Paul Kinnucan, "Computers that Think like Experts", *High Technol. Mag.*, Vol 71, January 1984, 30-42.
36. John F. Gilmore, Kirt Pulaski, and Chuck Howard, "A Comprehensive Evaluation of Expert System Tools", *Proc. Applications of Artificial Intelligence III*, SPIE Vol 635, April 1986, 2-16.
37. Fedrick Hayes-Roth, "Knowledge-Based Expert Systems", *IEEE Computer*, Vol 17, No. 10, October 1984, 263-273.

APPENDIXES

APPENDIX A

THE KAPPA.C PROGRAM FOR INITIATING KAPPA
AND REGISTERING FUNCTIONS

```

/*****
**                               KAPPA.c
*****/

/*****/
/* Copyright (c) MegaKnowledge 1988 */
/* VersionInfo: "%w %v %f" */
/* "RBP' 11 6-Mar-89,19:58:12" */
/* */
/*****/

/* KAPPA.c - Resident startup portion of Kappa */

/*****
** First include the files with the structures and definitions,
** as well as the global variable declarations.
*****/
#include "kappa.h"
#include "enhanced.c"

/* Function declarations */

/*****
** Main procedure, called by microsoft windows when
** KAPPA.EXE is activated.
*****/
int FAR PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow)
HANDLE hInstance, hPrevInstance;
LPSTR lpszCmdLine;
int cmdShow;
{
    if ( KappaInit( KERNEL, hInstance, hPrevInstance, lpszCmdLine, cmdShow )
        == FALSE )
        return FALSE;
    if ( KappaInit( DEVELOP, hInstance, hPrevInstance, lpszCmdLine, cmdShow )
        == FALSE )
        return FALSE;

    /*****/
    /* FOR THE USER: */
    /* - Please insert all your initialization code after this */
    /* comment and before the 'return' statement. */
    /* - At this point, all KAPPA specific information has */
    /* already been initialized. */
    /* - Please repeat appropriate initialization code in the */
    /* 'InitNewApplication'. This routine is called upon */
    /* selection of 'New' from the File menu of KAPPA. */
    /*****/
    /* Begin */

    Register_Function( "RunBackground", RunBackground, EVAL_ARGS, CAT_WND );
    Register_Function( "ObjectBreak", ObjectBreak, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectLength", ObjectLength, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectSegment", ObjectSegment, EVAL_ARGS, CAT_STR );

```

```

Register_Function( "ObjectPosition", ObjectPosition, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToUpper", ObjectToUpper, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToLower", ObjectToLower, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToProper", ObjectToProper, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToAnsi", ObjectToAnsi, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToChar", ObjectToChar, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToRadix", ObjectToRadix, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectSort", ObjectSort, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectIndex", ObjectIndex, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectbSearch", ObjectbSearch, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectInterpret", ObjectInterpret, EVAL_ARGS, CAT_STR );
Register_Function( "FileOpen", FileOpen, EVAL_ARGS, CAT_FILE );
Register_Function( "FileClose", FileClose, EVAL_ARGS, CAT_FILE );
Register_Function( "FileRead", FileRead, EVAL_ARGS, CAT_FILE );
Register_Function( "FileWrite", FileWrite, EVAL_ARGS, CAT_FILE );
Register_Function( "FilePosition", FilePosition, EVAL_ARGS, CAT_FILE );
Register_Function( "FileGoto", FileGoto, EVAL_ARGS, CAT_FILE );
Register_Function( "FileDir", FileDir, EVAL_ARGS, CAT_FILE );
Register_Function( "FileDelete", FileDelete, EVAL_ARGS, CAT_FILE );
Register_Function( "FileRename", FileRename, EVAL_ARGS, CAT_FILE );
Register_Function( "FileExec", FileExec, EVAL_ARGS, CAT_FILE );
Register_Function( "FilePrint", FilePrint, EVAL_ARGS, CAT_FILE );

/* End */
return KappaLoad( DEVELOP, lpszCmdLine );
}

/*****
** Initialization procedure, called by KAPPA when the New menu item
** is selected.
*****/
void InitNewApplication (void)
{
    /* Begin */

    Register_Function( "RunBackground", RunBackground, EVAL_ARGS, CAT_WND );
    Register_Function( "ObjectBreak", ObjectBreak, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectLength", ObjectLength, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectSegment", ObjectSegment, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectPosition", ObjectPosition, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectToUpper", ObjectToUpper, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectToLower", ObjectToLower, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectToProper", ObjectToProper, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectToAnsi", ObjectToAnsi, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectToChar", ObjectToChar, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectToRadix", ObjectToRadix, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectSort", ObjectSort, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectIndex", ObjectIndex, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectbSearch", ObjectbSearch, EVAL_ARGS, CAT_STR );
    Register_Function( "ObjectInterpret", ObjectInterpret, EVAL_ARGS, CAT_STR );
    Register_Function( "FileOpen", FileOpen, EVAL_ARGS, CAT_FILE );
    Register_Function( "FileClose", FileClose, EVAL_ARGS, CAT_FILE );
    Register_Function( "FileRead", FileRead, EVAL_ARGS, CAT_FILE );
    Register_Function( "FileWrite", FileWrite, EVAL_ARGS, CAT_FILE );

```

```

Register_Function( "FilePosition", FilePosition, EVAL_ARGS, CAT_FILE );
Register_Function( "FileGoto", FileGoto, EVAL_ARGS, CAT_FILE );
Register_Function( "FileDir", FileDir, EVAL_ARGS, CAT_FILE );
Register_Function( "FileDelete", FileDelete, EVAL_ARGS, CAT_FILE );
Register_Function( "FileRename", FileRename, EVAL_ARGS, CAT_FILE );
Register_Function( "FileExec", FileExec, EVAL_ARGS, CAT_FILE );
Register_Function( "FilePrint", FilePrint, EVAL_ARGS, CAT_FILE );

/* End */
}

/*****
** Initialization procedure, called by KAPPA when the OPEN menu item
** is selected.
*****/
void InitOpenApplication (void)
{
/* Begin */

Register_Function( "RunBackground", RunBackground, EVAL_ARGS, CAT_WND );
Register_Function( "ObjectBreak", ObjectBreak, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectLength", ObjectLength, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectSegment", ObjectSegment, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectPosition", ObjectPosition, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToUpper", ObjectToUpper, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToLower", ObjectToLower, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToProper", ObjectToProper, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToAnsi", ObjectToAnsi, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToChar", ObjectToChar, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectToRadix", ObjectToRadix, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectSort", ObjectSort, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectIndex", ObjectIndex, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectbSearch", ObjectbSearch, EVAL_ARGS, CAT_STR );
Register_Function( "ObjectInterpret", ObjectInterpret, EVAL_ARGS, CAT_STR );
Register_Function( "FileOpen", FileOpen, EVAL_ARGS, CAT_FILE );
Register_Function( "FileClose", FileClose, EVAL_ARGS, CAT_FILE );
Register_Function( "FileRead", FileRead, EVAL_ARGS, CAT_FILE );
Register_Function( "FileWrite", FileWrite, EVAL_ARGS, CAT_FILE );
Register_Function( "FilePosition", FilePosition, EVAL_ARGS, CAT_FILE );
Register_Function( "FileGoto", FileGoto, EVAL_ARGS, CAT_FILE );
Register_Function( "FileDir", FileDir, EVAL_ARGS, CAT_FILE );
Register_Function( "FileDelete", FileDelete, EVAL_ARGS, CAT_FILE );
Register_Function( "FileRename", FileRename, EVAL_ARGS, CAT_FILE );
Register_Function( "FileExec", FileExec, EVAL_ARGS, CAT_FILE );
Register_Function( "FilePrint", FilePrint, EVAL_ARGS, CAT_FILE );
RunModule();

/* End */
}
/*****
**
** END OF MAIN FILE
**
*****/

```

APPENDIX B

ALPHABETICAL REFERENCE FOR ALL ENHANCED

FUNCTIONS

FileClose	Format	<code>FileClose([number, ...])</code>
	Purpose	The files with the given file number(s) are closed.
	Arguments	File numbers of open files. If no arguments are given, file 0 is closed.
	Return Values	A TRUE on success else a FALSE is returned. An ERROR is returned if the number exceeds 19.
	Notes	Numbers can be in the range 0 to 19 inclusive. If no numbers are specified, the number is assumed to be 0.
	Example	<code>=>FileClose(0, 2, 4); TRUE</code>
FileDelete	Format	<code>FileDelete(object, slot);</code>
	Purpose	The file name denoted by the 'object, slot' pair is deleted. If the slot is multi-valued, all the files in the list are deleted.
	Arguments	A class (or instance) and slot name containing file name(s). One argument evaluating to a file name can also be given.
	Return Values	The function returns a TRUE on successful deletion of an existing file, else returns a FALSE.
	Notes	Files should not be open when deleting.
	Example	<code>=>FileDelete(Myjob.c); TRUE</code>

FileDir	Format	<code>FileDir(object, slot);</code>
	Purpose	Display and select file(s) in a directory. A standard Windows dialog box is displayed. If the slot is multi-valued, an entire selection of files pertaining to the given wild-card characters is stored in the slot.
	Arguments	A class (or instance) and slot name evaluating to a file name containing wild-card characters '*' and '?'. If the 'object, slot' pair is a list, the first element of the list should be the file name.
	Return Values	A TRUE on successfully obtaining the name(s) into 'object, slot'. The file name selected is returned if only one argument evaluating to a file name with wild-card characters is given. A FALSE is returned if no files are selected.
	Example	<code>=>FileDir("c:*.*");</code> <code>CONFIG.SYS // If CONFIG.SYS was selected from the dialog box.</code>
FileExec	Format	<code>FileExec(command-line[, mode]);</code>
	Purpose	The function executes any given command line containing a DOS or Windows executable file, and starts it in the given mode.
	Arguments	DOS or Windows executable file name with parameters if any. The mode can be NORMAL by default, to display a normal window in the foreground, BACKGROUND to display the window in the background, MINIMIZE to show it minimized and MAXIMIZE to show it maximized.
	Return Values	A TRUE if successful, else a FALSE if unable to execute.
	Notes	A DOS application runs full screen in any mode.
	Example	<code>=>FileExec("Notepad Myfile", MAXIMIZE);</code> <code>TRUE</code>

FileGoto	Format	<code>FileGoto(number, position);</code>
	Purpose	The function places the file pointer, of the file specified with number, at the given offset position in bytes from the beginning of the file.
	Arguments	The number signifies a file opened with that number. The position is the file pointers offset from the beginning of the file. A + or - following position increments or decrements the pointer position from its current position. A \$ following the position, would offset the pointer from the end of the file.
	Return Values	A TRUE is returned on success, a FALSE on failure.
	Notes	If arguments are not specified, a default of 0 is assumed for both arguments. This function also works with files opened with <code>OpenReadFile()</code> and accessed with <code>ReadWord()</code> .
	Example	<code>=>FileGoto(); TRUE // Sets pointer to the beginning of file 0</code>
FileOpen	Format	<code>FileOpen(number, file-name[, mode]);</code>
	Purpose	Open a binary text file, for reading, writing, modifying or appending.
	Arguments	The number is a file number greater or equal to 0 which is assigned as the file pointer. A file name can be given directly or an object:slot can contain the file name. The modes can be read, write, append or modify.
	Return Values	If the number has already been assigned to a file, the function returns a FALSE else if successful, returns the name of the file.
	Notes	The mode is read by default and the number is 0, if only one argument is specified. A file opened with the number 0 or without a number specified, is equivalent to using the already built in <code>OpenReadFile</code> function, except that it is opened in the binary, rather than text, mode. Using number 1 is the same as the <code>OpenWriteFile</code> function. Wild-card characters given in the file name, cause WINDOWS to display a standard dialog box for selection of a file.
	Example	<code>FileOpen(0, "c:myfile", modify); C:MYFILE // The file-name converted to upper-case is returned.</code>

FilePosition	Format	FilePosition(number);
	Purpose	This function returns the current position of the file pointer, in bytes, for a given file.
	Arguments	The number is a file number of an open file.
	Return Values	It returns the new position of the file in bytes, from the beginning of the file. It returns an ERROR if unsuccessful.
	Notes	An ERROR is returned if the function is used without first opening the relevant file. If no argument is given, the file number is taken to be 0. It also returns the position of a file opened using OpenReadFile() and read using ReadWord().
	Example	<code>=>FilePosition(3);</code> <code>10 // If the file pointer is at the tenth byte.</code>
FilePrint	Format	FilePrint(filename[, pt.size[, options[, font[, printer]]]]);
	Purpose	The given file is printed on the default Windows printer. An optional last parameter can be defined to print on any printer installed under Windows.
	Arguments	The filename can be any existing file. A wild-card argument displays a Windows dialog to choose a file. The default point size for printing is 12, but can be overridden by the second argument. The options consist of a string of characters wherein each character sets a different option and the characters can be arranged in any order. The different character options available are: n - normal(default), b - bold, i - italic, u - underline, v - variable font, s - script font, t - no top or left margins, r[n] - set right margin to a maximum of n characters per line, h - header(file name and current date), p - page numbers, l - line numbers and m - multi-column output. An optional font name such as Courier, Helv or Symbol can be supplied as the font argument. If a valid printer name is supplied, output is directed to that particular printer else output is sent to the default Windows printer. A wild-card argument displays a menu to select from a maximum of ten printers.
	Return Values	The function returns the name of the file being printed, if successful, else a FALSE is returned.
	Notes	If no arguments are given or the DOS file 'NUL' is declared, the number of lines in a page is returned and no other action taken.
	Example	<code>=>FilePrint("c:\config.sys", 10, bhpr72v, NULL, ?);</code> <code>C:\CONFIG.SYS //The file is printed in bold with a header, page numbers, a maximum of 72 characters per line and the default variable font of point size 10. The printer is selected from a menu.</code>

FileRead	Format	FileRead(number, comment-delimiter, field-delimiter, record-delimiter[, object, slot1, object, slot2, ...]);
	Purpose	Read an ASCII text file containing remarks, fields and records.
	Arguments	The number is the file number with which the file was opened. The comment delimiter disregards any records that begin with the comment character. The field delimiter breaks a record into fields whenever it comes across the field string. The record delimiter denotes the termination string for an entire record. Only the first character is taken for the comment delimiter whereas a string can be given for the field and record delimiters. The string can be a C format string where a \n would indicate the new-line character. The 'object, slot' pairs are where each field value is stored. These can denote lists.
	Return Values	If no slot names are given, a single record which is read, is returned. If the slots are single-valued, one valid record is read and each field in that record is stored in its corresponding slot. The last field or record is returned. If the slots are multi-valued, a maximum of 100 records are read and stored into the corresponding lists after each record is broken into fields. Fields or slots in excess are made NULL. The function returns the actual number of records read. A 0 indicates that no records were read and a FALSE indicates the end of file.
	Notes	This function also yields to other Window applications, so that the read operation can be carried out in the background. If the Object Browser or the Knowledge Tools windows are open, and new classes or instances are being created along with this function, these windows are updated to show the current status. This can slow down the actual read process. If speed is required, these windows should be closed. If the record delimiter is NULL, a record of 200 characters is returned.
	Example	=>FileRead(0, *, NULL, "\n"); THE FIRST LINE NOT BEGINNING WITH AN * IS RETURNED.
FileRename	Format	FileRename(object, slot1, object, slot2);
	Purpose	To rename an existing file or a group of files. The file name given in slot1 is renamed to the one given in slot2. If slots are multi-valued, a one to one correspondence is assumed between the different files.
	Arguments	The 'object, slots' should contain valid file names. Arguments evaluating to 2 strings can also be given, as shown in example below.
	Return Values	The name of the file in the list, where a renaming error occurred, is returned, else a TRUE is returned on success.
	Example	=>FileRename(oldname.txt, Global:NewName); TRUE

FileWrite	Format	FileWrite(number, field-delimiter, record-delimiter, object, slot1, [object, slot2,...]);
	Purpose	Write to an ASCII text file with data separated by field and record delimiters.
	Arguments	The number denotes the file number under which a file has been opened for writing. The field delimiter is added after writing each slot value and the record delimiter is added after the last slot is written. The slots can be single valued in which case one record is written, or they can be multi-valued in which case the maximum number of records written to the file, is the length of the list.
	Return Values	The record itself is returned if single, else the number of records written is returned or in case of error, an error message is displayed
	Notes	This function has the same ability as the ReadWord function to yield control to other Window processes. It can be used in conjunction with the FormatValue() function to format output text.
	Example	=>FileWrite(0, "", "\n", "This is a sample line."); This is a sample line.
	ObjectBreak	Format
	Purpose	This function is useful in breaking a string into tokens, each separated by any one character in the pattern string.
	Arguments	The source string exists in slot1. The pattern to match and then break the source into tokens is the pattern string. Optional target 'object, slot' pairs are where these tokens are placed after breaking them. The target can also be declared within quotes as "object:slot" pairs. The Pattern String itself can be given directly within quotes or in an object:slot pair. It can also contain special characters using the back-slash as the first character, like \n.
	Return Values	The function takes the value of an 'object, slot' pair in KAPPA and returns the first sub-string or token terminated by any of the characters present in the pattern string. If the optional target 'object, slot' pairs are defined, the function returns a TRUE on success.
	Notes	If the 'object, slot' pairs are lists, the function breaks each string in the list, according to the pattern, into tokens and places each one into the corresponding target list. If there are less target lists than tokens, it ignores the rest but returns 'Less Slots', whereas if there are less tokens, it leaves the excess slots empty.
	Example	=>ObjectBreak("744-1998", "-"); 744

ObjectbSearch	Format	ObjectbSearch(object, slot1, key[, object, slot2[, sec.key, object, slot3]]);
	Purpose	A binary search is performed, given a primary key and an optional secondary key, on a list which has been sorted or indexed.
	Arguments	'object, slot1' denote a multi-valued list which is either sorted or indexed. The key is the primary key on which search is conducted. If the list is indexed, another multi-valued list denoted by 'object, slot2' and containing the original unsorted list has to be defined. With indexed lists, a secondary key can also be specified along with a secondary list contained in 'object, slot3'. If two or more items with the same primary key are found, the secondary key is matched with items in the secondary list.
	Return Values	Returns the position of the key as found in the sorted or indexed list.
	Notes	A sort or index has to be performed before this search.
	Example	=>ObjectbSearch(Global, IndexedList, Tom, Global, OriginalList); 20 // The 20th element in the indexed list contains the position of Tom in the original list.
ObjectIndex	Format	ObjectIndex(object, slot1, object, slot2[, object, slot3]);
	Purpose	Index the values contained in a list and place their positions, in ascending order, into another list.
	Arguments	'object, slot1' is a multi-valued slot containing the original values of strings or numbers. 'object, slot2' is the destination slot where the list of index values is placed. An optional secondary list on which the indexing is performed is defined by 'object, slot3'.
	Return Values	A TRUE on success else an appropriate error message is returned.
	Notes	The indexing provides an alternate method to sorting, specially when the original list does not need a physical change or when more than one, parallel lists are used.
	Example	=>ObjectIndex(Global, AllNames, Global, IndexedNames); TRUE

ObjectInterpret	Format	ObjectInterpret([LIST,] expr1[, expr2, ...]);
	Purpose	Concatenate, interpret and execute valid KAL commands given in one or more expressions.
	Arguments	One or more expressions given as arguments are concatenated together to form a KAL statement. If the KAL expression returns a list, the key-word LIST has to be specified as the first argument.
	Return Values	If valid, the KAL expression is executed and its value returned, else a FALSE is returned.
	Notes	expr1, expr2 ... when concatenated together should be a valid KAL expression including the end semicolon.
	Example	=>For x [1 3] AppendToList(Global:Lst, ObjectInterpret("LengthList(Global:", Lst#x, ");")); LIST: 100 50 200 // Returns the length of lists Global:List1 thru Lst3
ObjectLength	Format	ObjectLength(object, slot);
	Purpose	To find the length of a string, list or class.
	Arguments	The 'object, slot' pair can be single-valued, containing a string or a multi-valued list. A string or class name can also be passed as one single argument.
	Return Values	It returns the length of the string specified. If the 'object, slot' pair is a list, the length of the list is returned. A 0 is returned if the 'object, slot' pair is empty or NULL. If a class name is passed directly, as a single argument, the number of instances for that particular class are returned.
	Notes	A class name can be passed in a slot by using object:slot which is evaluated before it is passed.
	Example	ObjectLength(Root); 1 // The number of instances contained in Root.

ObjectPosition	Format	ObjectPosition(object, slot, string-segment, start-position);
	Purpose	To find the overall position of a sub-string or element in another string or list, optionally beginning from the given position.
	Arguments	The 'object, slot' can be single or multi-valued containing either the string or the list to be searched. The string segment is a sub-string or element in the list to search for. If the 'object, slot' is specified as a single argument such as 'object:slot', the start position is optional.
	Return Values	The position of the sub-string in the string or the position of the element in a list is returned. Starting from the optional start position can make it skip sub-strings occurring before this position. A 0 is returned if the segment is not found.
	Notes	The 'object, slot' can also be specified as a single value evaluating to a string or class name. If a class is specified, and the string-segment is an Instance name, the position of the Instance in the Class is returned.
	Example	<pre>=>ObjectPosition("405-744-1998", -, 5); 8 // The position of the second hyphen in a telephone number.</pre>
ObjectSegment	Format	ObjectSegment(object, slot, start-position, length);
	Purpose	Finds a segment of a specified string given the beginning position and the length required.
	Arguments	The 'object, slot' can either be single or multi-valued. It can also be specified as one single string or class name. The start-position is the position from which the segment is extracted. It is the position of a character in a single-valued slot or the element position in a multi-valued list. The length is the number of characters to copy for a single-valued slot or the number of elements for a multi-valued slot.
	Return Values	Returns the extracted segment if 'object, slot' is specified as a single string. A TRUE is returned if the object and slot name are given separately. The new value replaces the old if single-valued. If a list, it is replaced by a sub-list starting with the first element given in the start-position with length number of elements. If a class name is passed directly to the function, a sub-list of the instances beginning at the start-position and length in number is returned.
	Notes	The class name in a slot can be passed directly using object:slot.
	Example	<pre>=>ObjectSegment("(405) 744-1998", 2, 3); 405 // The area code starting at position 2 and length 3 is returned.</pre>

ObjectSort	Format	<code>ObjectSort(object, slot);</code>
	Purpose	Sorts a multi-valued list and places the values in lexicographic order using the Heap-Sort algorithm.
	Arguments	The 'object, slot' denotes any multi-valued slot can be given as an argument to this function.
	Return Values	The sorted list replaces the original list. A TRUE is returned upon completion.
	Notes	A single value can also be passed, in which case, this value should evaluate to an 'object:slot' pair.
	Example	<code>=>ObjectSort("Global:SortedList"); TRUE // The elements in SortedList are sorted and replaced.</code>
ObjectToAnsi	Format	<code>ObjectToAnsi(object, slot);</code>
	Purpose	To obtain the Windows ANSI numeric equivalent for any character.
	Arguments	An 'object, slot' pair evaluating to a string or list. The ANSI equivalent of the first character in this string or the first character of every element in the list is found.
	Return Values	If a single string argument is given the ANSI value is returned directly, else a TRUE is returned after setting the slot value to the ANSI numeric equivalent(s).
	Notes	The original string or list is lost when the 'object, slot' pair is passed. The Windows ANSI values differ from the DOS ASCII values for the extended character set.
	Example	<code>=>ObjectToAnsi(A); 65 // The ANSI equivalent of the letter A.</code>

ObjectToChar	Format	<code>ObjectToChar(Object, Slot);</code>
	Purpose	Converts a number to its equivalent Windows ANSI character format.
	Arguments	An 'object, slot' pair evaluating to a number or list of numbers. The ANSI equivalent character of these numbers is found.
	Return Values	If a single numeric argument is given the equivalent ANSI character is returned directly, else a TRUE is returned after setting the slot value to the ANSI character equivalent(s).
	Notes	The original number or list is lost when the 'object, slot' pair is passed. The Windows ANSI values differ from the DOS ASCII values for the extended character set.
	Example	<code>=>ObjectToChar(65); A // The ANSI character equivalent of 65.</code>
ObjectToLower	Format	<code>ObjectToLower(Object, Slot);</code>
	Purpose	Converts strings, list of strings, classes or instances to lowercase.
	Arguments	An 'object, slot' denoting a string or a list of strings. If passed as a single argument, it can be either a string, class or instance name.
	Return Values	If a single string argument is given, it is converted into lowercase letters. If this is a class or instance name, the class or instance is converted to lowercase. Otherwise, a TRUE is returned after converting the slot value(s) to lowercase.
	Notes	The original string(s) are lost when the 'object, slot' pair is passed. The single value can be passed in a slot by specifying it as object:slot.
	Example	<code>=>ObjectToLower(LOWER); lower // The string is converted to lowercase.</code>

ObjectToProper	Format	<code>ObjectToProper(Object, Slot);</code>
	Purpose	Converts strings, list of strings, classes or instances to lowercase with the first character capitalized.
	Arguments	An 'object, slot' denoting a string or a list of strings. If passed as a single argument, it can be either a string, class or instance name.
	Return Values	If a single string argument is given, it is converted into the proper-noun form with the first character capitalized and all the others in lowercase. If this is a class or instance name, the class or instance is also similarly converted. Otherwise, a TRUE is returned after converting the slot value(s).
	Notes	The original string(s) are lost when the 'object, slot' pair is passed. The single value can be passed in a slot by specifying it as object:slot.
	Example	<code>=>ObjectToLower(PROPER);</code> <code>Proper // The string is converted to the proper-noun form.</code>
ObjectToRadix	Format	<code>ObjectToRadix(original-value, original-base, new-base);</code>
	Purpose	Convert a value in the given original base to its equivalent new base.
	Arguments	The original value is a number in the given base. The original base contains the base of the original value in decimal format. The new base contains the base to which the original value is to be converted to, also in decimal format.
	Return Values	The original value after being converted to the new base is returned. The base values can range between 2 and 36. A FALSE is reported on failure.
	Notes	Lists are not supported. Return values need not be all numeric and can contain letters from A to Z depending on the base.
	Example	<code>=>ObjectToRadix(FAB, 16, 2);</code> <code>111110101011 // The hex value of 4011 is converted to binary form.</code>

ObjectToUpper	Format	<code>ObjectToUpper(Object, Slot);</code>
	Purpose	Converts strings, list of strings, classes or instances to uppercase.
	Arguments	An 'object, slot' denoting a string or a list of strings. If passed as a single argument, it can be either a string, class or instance name.
	Return Values	If a single string argument is given, it is converted into uppercase letters. If this is a class or instance name, the class or instance is converted to uppercase. Otherwise, a TRUE is returned after converting the slot value(s) to uppercase.
	Notes	The original string(s) are lost when the 'object, slot' pair is passed. The single value can be passed in a slot by specifying it as object:slot.
	Example	<code>=>ObjectToLower(upper); UPPER // The string is converted to uppercase.</code>
RunBackground	Format	<code>RunBackground();</code>
	Purpose	To be able to run other applications while KAPPA is processing.
	Arguments	None.
	Return Values	It always returns a TRUE.
	Notes	This statement should be included within the main processing loop within which, while processing, other applications need to be run. This function allows the current application to yield to Windows so that other applications demanding attention from Windows can be processed. After processing other application messages, Windows gives back the control to this application to continue processing. The effect is of being able to run the current process in the background.
	Example	<code>RunBackground(); TRUE // Allows Windows to take over control.</code>

VITA 

Rohinton N. Mistry
Candidate for the Degree of
Master of Science

Thesis: ENHANCEMENTS TO KAPPA, AN OBJECT ORIENTED EXPERT
SYSTEM SHELL

Major Field: Computer Science

Biographical:

Personal Data: Born in Bombay, India, August 26, 1963, the son of Noshir and Rashna Mistry.

Education: Graduated from Hyderabad Public School, Begumpet, Hyderabad, India, in March 1981; received Bachelor of Science Degree in Math, Physics and Chemistry from Osmania University, Hyderabad, India, in 1985; received Post Graduate Diploma in Software Technology from Bureau of Data Processing Systems, Secunderabad, India, in May 1986; completed requirements for the Master of Science degree at Oklahoma State University in May 1992.

Professional Experience: Programming Instructor, Bureau of Data Processing Systems, January 1987 to December 1989; Graduate Research Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, January 1991 to May 1992.