1993-03

Design and implementation of an interface editor for the Amadeus multi-relational database front-end system.

Hargrove, James Phillip.

Monterey, California: Naval Postgraduate School

https://hdl.handle.net/10945/24147

Approved for public release; distribution is unlimited

Design and Implementation of an Interface Editor for the Amadeus Multi-Relational Database Front-end System

by
James Phillip Hargrove
Lieutenant Commander, United States Navy
BA, University of California, Berkeley, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March, 1993

UNCLASSIFIED
JRITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE		
ꞌERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION ɔmputer Science Dept. ıval Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) onterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NC

TITLE (Include Security Classification)
:SIGN AND IMPLEMENTATION OF AN INTERFACE EDITOR FOR THE AMADEUS MULTI-RELATIONAL DATABASE
ꝜONT-END SYSTEM (U)

PERSONAL AUTHOR(S)
CDR James Phillip Hargrove, USN

a TYPE OF REPORT aster's Thesis	13b. TIME COVERED FROM 02/91 03/93	14. DATE OF REPORT (Year, Month, Day) 930325	15. PAGE COUNT 286

ɪOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or
ısition of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	OBJECT-ORIENTED PROGRAMMING, USER INTERFACE DESIGN, PROGRAPH VISUAL PROGRAMMING, FORM-BASED INTERFACE, DATABASE SYSTEMS

ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis extends the Graphical User Interface of a prototype multi-relational database front-end system, callec
madeus. System enhancements are realized through the application of Object-Oriented Programming (OOP) anc
uman-Computer Interface (HCI) design principles. Knowledge gained from each topic has been incorporated intc
e design and implementation of a Form-based interface for database data entry and display.

The focus of this thesis is divided between two issues: the development of a set of tools for creating and using *Forn*
ɔjects; and the design of the *Form* object itself. Form creation is accomplished using an application program callec
c *Interface Editor module*. The Interface Editor is one of six modules which, together, comprise the Amadeus sys
m. Form manipulation occurs in a second application which implements basic program methods for controlling dat
ıtry and display processes.

Design and implementation of this thesis was accomplished using the Prograph programming language and devel
ɔment environment, which provided a basic set of system classes essential to the implementation of the Form objec
ıd Graphical User Interfaces.

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
a NAME OF RESPONSIBLE INDIVIDUAL ofessor C. Thomas Wu	22b. TELEPHONE (Include Area Code) (408) 656-2174	22c. OFFICE SYMBOL CS/Wu

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ABSTRACT

This thesis extends the Graphical User Interface of a prototype multi-relational database front-end system, called Amadeus. System enhancements are realized through the application of Object-Oriented Programming (OOP) and Human-Computer Interface (HCI) design principles. Knowledge gained from each topic has been incorporated into the design and implementation of a Form-based interface for database data entry and display.

The focus of this thesis is divided between two issues: the development of a set of tools for creating and using *Form* objects; and the design of the *Form* object itself. Form creation is accomplished using an application program called the *Interface Editor module*. The Interface Editor is one of six modules which, together, comprise the Amadeus system. Form manipulation occurs in a second application which implements basic program methods for controlling data entry and display processes.

Design and implementation of this thesis was accomplished using the Prograph programming language and development environment, which provided a basic set of system classes essential to the implementation of the Form object and the Graphical User Interfaces developed for this thesis.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank the following persons for their advice, assistance and encouragement during the course of this thesis research: Professor C. Thomas Wu, LCDR John A. Daley, USN, LCDR Kim L. Kotlar, USN, LT Robert S. Lovejoy, USN, LCDR Andy Melton, USN and 1LT Turgay Cince, Turkish Army.

I. INTRODUCTION

This thesis investigates two separate topics, object-oriented programming and Human-Computer Interface (HCI) design. Knowledge gained from each is applied to the design of a Form-based interface for database data entry and display to be used in conjunction with a new relational database query language called *Dataflow Query Language (DFQL)*. Central to the Form-based interface is the design of a *Form object* which is an abstract representation of a computer generated *Form* (and its component parts). Forms of various types are common in everyday life. Examples include income tax forms, questionnaires, job applications, invoices and order forms. Form-based interfaces are popular for database applications since they extend the paper form by creating a new computer metaphor for these familiar objects. Users are thus able to manipulate form images through a Graphical User Interface (GUI) in a similar manner as they would a paper form. This ability greatly assists user interaction with the system, since users see a display of all related fields and have a feeling of control over the data entry process [Schneiderman92 page 71; pages 132–133]. Additionally, few instructions are necessary since the computer-displayed form so closely resembles familiar paper forms [Schneiderman92 page 132].

The Form-based interface discussed above actually consists of two separate applications: The *Interface Editor* module and the *Form Use* application program. The Interface Editor is one of six modules which, together, comprise a prototype multi-relational database front-end system called *Amadeus*.[1] The purpose of the Interface Editor module is to provide users with a tool for creating and editing custom data entry and display Forms for use by Amadeus' Query Editor module. The Query Editor uses the DFQL query language for database manipulation, providing an "improved interface to the relational

[1] Amadeus and DFQL will be discussed further in Chapter II.

model of database management" [Clark91 p. 25]. Forms are used in conjunction with DFQL queries to make data entry and display more efficient for the system user.

The Interface Editor module is a stand-alone application program which is intended to be used separately from the Amadeus database definition and manipulation modules. This separation results in important advantages:

1. First, a clear distinction between Form creation and use is realized, simplifying the user interfaces for both the Interface Editor and other Amadeus modules.
2. Secondly, separating the Interface Editor functions from the remainder of the Amadeus modules keeps application size to a minimum while supporting general object-oriented programming goals by assisting in modular program development, testing and debugging.

The Form Use application program is used to validate the Interface Editor's Form creation capability, and incorporates control functions for displaying and controlling Form objects used for data entry and display. The component parts of the Form Use application will ultimately become part the Amadeus Query Editor module, which contains query editing and execution methods for the entire system. The Query Editor module is currently the topic of another thesis project, and integration of the Form Use functions will be accomplished as part of that work.

All design and programming for this thesis was accomplished using the Prograph™[2] programming language and development environment. Prograph is a visual language based on the dataflow paradigm. Since Prograph uses the Macintosh ROM-based toolbox and operating system managers, the Interface Editor and Form Use interfaces take on a distinctive Macintosh "look and feel".

Prograph language features relating to program development and source code listings for this thesis are discussed in Chapter II and Appendix B. Relevant Human Computer Interface design and implementation issues are discussed in Chapter III. The design and

[2] Prograph is a trademark of The Gunakara Sun Systems, Ltd.

implementation of the Form object, Interface Editor and Form Use application are detailed in Chapter IV. Chapter V provides a summary of this thesis, complete with conclusions and recommendations for further research. Source code listings for the Interface Editor and Form Use applications are contained in Appendix E.

II. TERMINOLOGY, BACKGROUND AND BASIC CONCEPTS

This chapter provides background information on the Prograph programming language and development environment, DFQL and the Amadeus system. Each of these concepts will be discussed briefly. An introduction to the Prograph language will then be presented to assist in the interpretation of code listings found in Appendix E. Finally, DFQL and Amadeus will be described. Additional information on these subjects may be found in Appendices B and C.

A. HIGH-LEVEL PROGRAMMING LANGUAGES

One way of classifying a programming language is by the language's level of *abstraction*. A language is considered higher-level than another if it can express a program with less detail than the second language. This means that a high-level language enables a programmer to concentrate more on *what* is to be done and less on *how* to do it [MacLennan87 p. 485-486]. The authors of Prograph assert that it is a "very high level" language. This implies that Prograph is about as far removed from machine language as is practical, and is more abstract than traditional high-level programming languages such as Ada, Fortran or Pascal.

B. VISUAL PROGRAMMING

S. K. Chang describes two general categories of visual languages: those that process visual information and those that make use of visual expression (known as *Visual Programming Languages*). Languages that are used to process visual information are usually traditional, linear languages that have been specifically enhanced to handle visual information or objects. Visual Programming Languages, on the other hand, deal with objects which are not normally expressed visually, but which are *themselves* visual in nature. Prograph is a example of a visual, very high-level programming language.

C. OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) differs from traditional procedural programming by the way in which data and action are treated. In procedural programming, data and action are treated separately. Typically, data structures are first defined, and then a set of procedures are developed to manipulate the data structures. With OOP, however, action and data are closely coupled (i.e., data and the actions associated with the data are defined together).

This section discusses the general characteristics of Object-Oriented Programming Languages (OOPLs), and defines basic terms and concepts.

1. Data Hiding (Encapsulation)

Data hiding is a process by which data can only be accessed by code which is specifically associated with the data. Data hiding is also called *encapsulation* because data and its associated code are placed together in a package or "capsule". Encapsulation is an important feature of OOP languages, and is also incorporated into certain procedural languages such as Modula-2 and Ada. By encapsulating code and data, the programmer guarantees that portions of a program which do not relate to specific data remain separate, and cannot access that data.

Data hiding aids in modular program construction, since details of the inner workings of a package are not required by anything outside of the package. When applied properly, modular program design enables packages (modules) to be added, modified or removed from a program with no impact on other modules. This is generally not the case with procedural languages which do not support encapsulation. In such languages, making a change to one part of a program may impact other parts of the program, producing a "ripple" effect of program and data dependencies. Encapsulation has been implemented in the design of both programs developed for this thesis. Portions of the Interface Editor have been incorporated into the Form Use application, which will, itself, ultimately be incorporated into the Amadeus Query Editor.

2. Classes and Objects

An *object* is an entity that contains data and an associated set of actions that operate on the data. Every object belongs to a *class*, which defines the implementation of a particular kind of object. An individual object of a class is referred to as an *instance* of the class. Classes can be thought of as templates for creating objects.

In object-oriented programming, when a class is created, *attributes* and *methods* are defined for the class. Attributes are a type of place holder for a specific value. Objects may have zero, one or many attributes. Methods represent known behaviors of instances of a class, and are roughly analogous to subroutines in more traditional languages. [TGS90b p. 146, TGS90c p. 4, Symantec91 p. 19-20] The objects, classes and methods developed as part of this thesis are discussed in Chapter IV.

3. Messages

In object-oriented programming, objects communicate with other objects by sending and receiving *messages*. Messages are analogous to sub-routine calls in a procedural programming language, and are used to tell an object to perform a specific action.

4. Inheritance

New classes can be defined in terms of existing classes through a process called *inheritance*. The new class is called a *subclass* or *child*, and the existing class (from which the subclass was defined) is called a *superclass* or the *parent* class. A subclass inherits all of the attributes and methods of its parent class. The parent, in turn, may have inherited attributes and methods from *its* parent class. A class inherits the combined attributes and methods of its ancestors. However, a subclass does not actually copy this information. Rather, it refers to the information as a parent class or superclass [Smith91 p. 65].

Subclasses are normally created when a new class is needed which differs slightly from an existing class. This is possible because a subclass can have its own attributes and methods not contained in the parent class. Prograph supplies a core of system-defined classes that describe Macintosh interface data structures such as menus, windows and window items. Subclasses of these system classes are developed by the programmer to define specific program actions. Figure 1 shows the Prograph System Class Hierarchy.

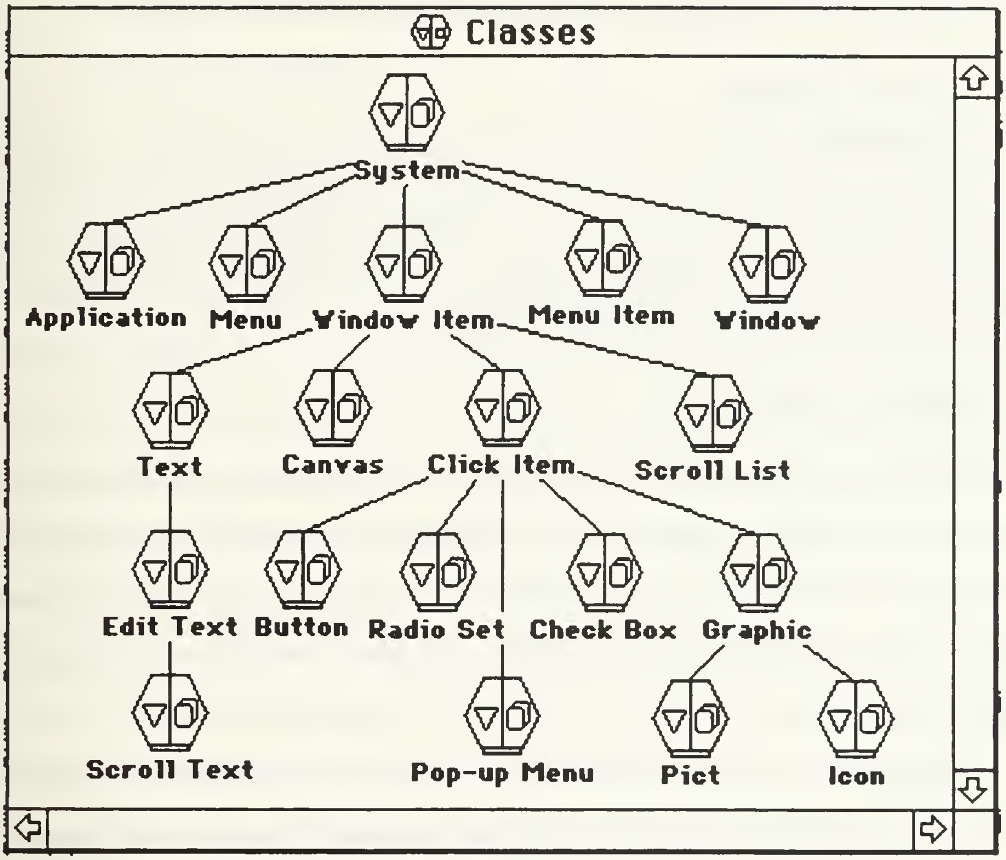


Figure 1: The Prograph System Class Hierarchy

5. Polymorphism

Simply stated, *polymorphism* allows the same message to be sent to objects of different classes. Each object invokes a method appropriate for its particular class [Smith91 p. 8, 109; TGS90b, p. 93]. Using the Macintosh Finder as an example, the effect of the *Open* command depends on the icon which has been selected. If a folder icon is selected, a folder is opened. If an application icon is selected, the application is opened. In effect, an Open message is being sent to various Finder objects, with each object applying its own Open method, as appropriate. [Symantec91 p. 22-23]

D. DATAFLOW DIAGRAMS & DATAFLOW PROGRAMMING

1. Dataflow Diagrams

A dataflow diagram is a modeling tool that is used extensively in operations research and computer science as an aid in systems analysis and design. A dataflow diagram makes use of distinct graphical symbols to convey meaning, is inherently visually oriented and is closely related to the directed graph.

2. Dataflow Programming

Dataflow programming is a natural extension of the dataflow diagram. A dataflow program is itself a dataflow diagram, so this type of programming allows the construction of two-dimensional graphical dataflow models that are directly translatable into computer executable instructions. Thus, a dataflow program is, simultaneously, a system model and an executable program.

Dataflow programming does not impose a specific ordering on program events. Rather, data can be thought of as actively flowing throughout a program, triggering events in a non-sequential manner as various data dependencies are satisfied, and corresponding program instructions are executed. This active flow of data implies more than one instruction can be evaluated simultaneously, making dataflow programming inherently concurrent. [TGS90b]

E. PROGRAPH

Prograph is an object-oriented, graphic programming language and development environment that supports the entire software design process. Prograph consists of the following integrated components [TGS90a p. 1]:

(1) pictorial language,

(2) graphic editor/interpreter development environment,

(3) Application Builder object-oriented interface building toolkit, and

(4) 680x0 code compiler

Prograph's visual environment makes it different from other programming languages. The language syntax is entirely pictorial, with text used only for comments and assigning names to objects, classes, methods, etc. Code for Prograph applications are composed of a series of dataflow diagrams consisting of operations (represented by icons) connected by datalinks. Data flows into an operation at a *terminal* node located at the top of the operation icon, and flows out of an operation from a *root* node located at the bottom of the operation icon. Datalinks provide the paths along which data flows into and out of operations. Dataflow diagrams in Prograph are displayed in *case windows*. Figure 2 shows a typical case window.

Prograph, as implemented on the uni-processor Macintosh, does not support concurrency. Although the Macintosh's uni-processor architecture limits program execution to a single instruction at a time, there is no sequential ordering placed on the execution of operations in a Prograph program. Therefore, the overall character of a dataflow program is preserved.

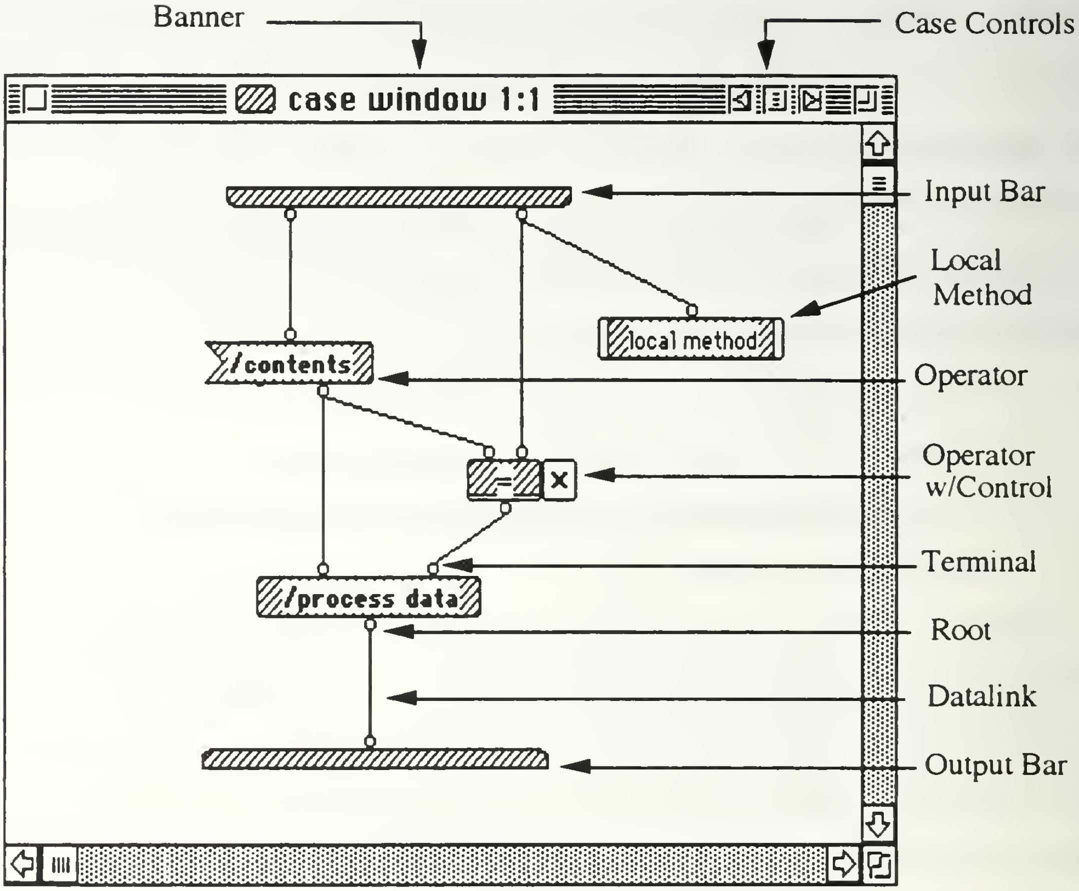


Figure 2: Prograph Case Window

1. Application Builder

Prograph incorporates an object-oriented, graphical application-interface toolkit (the *Application Builder*) which seamlessly integrates the Prograph development environment with the Macintosh ROM-based Toolbox[1] and operating system managers [TGS90a, TGS90b]. Traditional User Interface Management Systems or Interface Toolkits, which separate the user interface from the application program, require a complex, sequential edit-link-compile-run cycle whenever changes are made to an application.

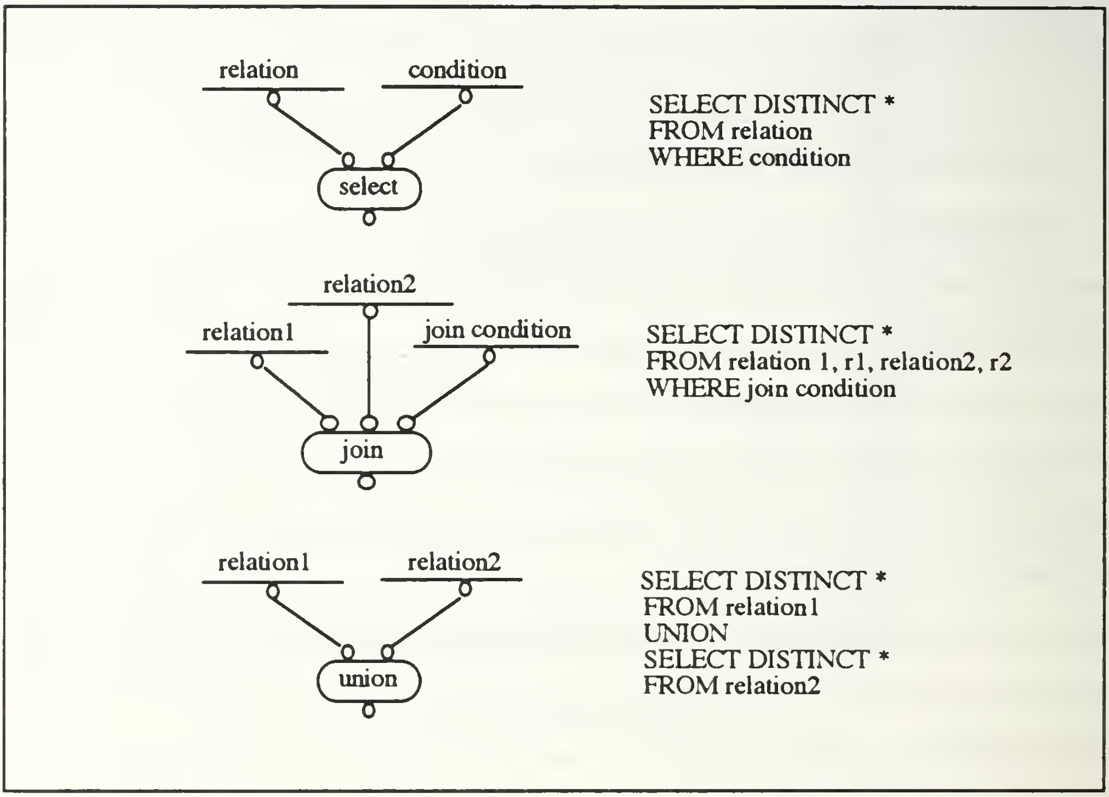
[1] The Macintosh Toolbox is a collection of low-level routines that implement the Macintosh operating system and user interface. See also [Apple85].

However, since Prograph's Application Builder facilities are an extension of the editor and interpreter, a dynamic run-edit-design-run program development cycle is employed which significantly reduces application development time. Thus, the Application Builder provides a seamless integration between the processes for developing user interfaces and the classes and methods that implement the fundamental behavior of the application itself [TGS90b p. 215-216].

F. DATAFLOW QUERY LANGUAGE

The Dataflow Query Language (DFQL) is a visual, relational algebra which is used to manipulate relational databases. Like Prograph, DFQL is a dataflow language, possessing sufficient expressive power and functionality to allow a user to easily express database queries. DFQL operators are constructed using three basic components: input nodes, a body and output nodes. These components correspond to the Prograph constructs terminal, node and root, respectively, and possess the same underlying principles and characteristics as their Prograph counterparts. Figure 3 compares three DFQL primitive operators with their equivalent *Structured Query Language (SQL)* queries. SQL has become the *de facto* industry standard query language for relational Database Management Systems (DBMS'). However, SQL has problems with both its design and implementation [Codd88a, pages 45-48, Codd88b pages 71-74, Codd90]. DFQL was developed to "allow users to achieve the maximum utility from the relational model" by providing an "improved interface to the relational model of database management" [Clark91 page1; page 25].

DFQL operators can be grouped into two basic categories: *primitive* and *user-defined*. Primitive operators have a one-to-one correspondence with methods in the implementation language of the interpreter. User-defined operators are created from primitive operators and possibly other user-defined operators which have been created previously. This provides a great deal of freedom and flexibility when constructing queries, since commonly used queries can be combined into compact, user-defined queries, eliminating the need to reconstruct them each time they are needed. Figure 4 shows a sample DFQL query, involving both primitive and user-defined operators. Additional database concepts are briefly discussed in Appendix C.



```
SELECT DISTINCT *
FROM relation
WHERE condition
```

```
SELECT DISTINCT *
FROM relation 1, r1, relation2, r2
WHERE join condition
```

```
SELECT DISTINCT *
FROM relation1
UNION
SELECT DISTINCT *
FROM relation2
```

Figure 3: DFQL Primitive Operators and Their Corresponding SQL Queries

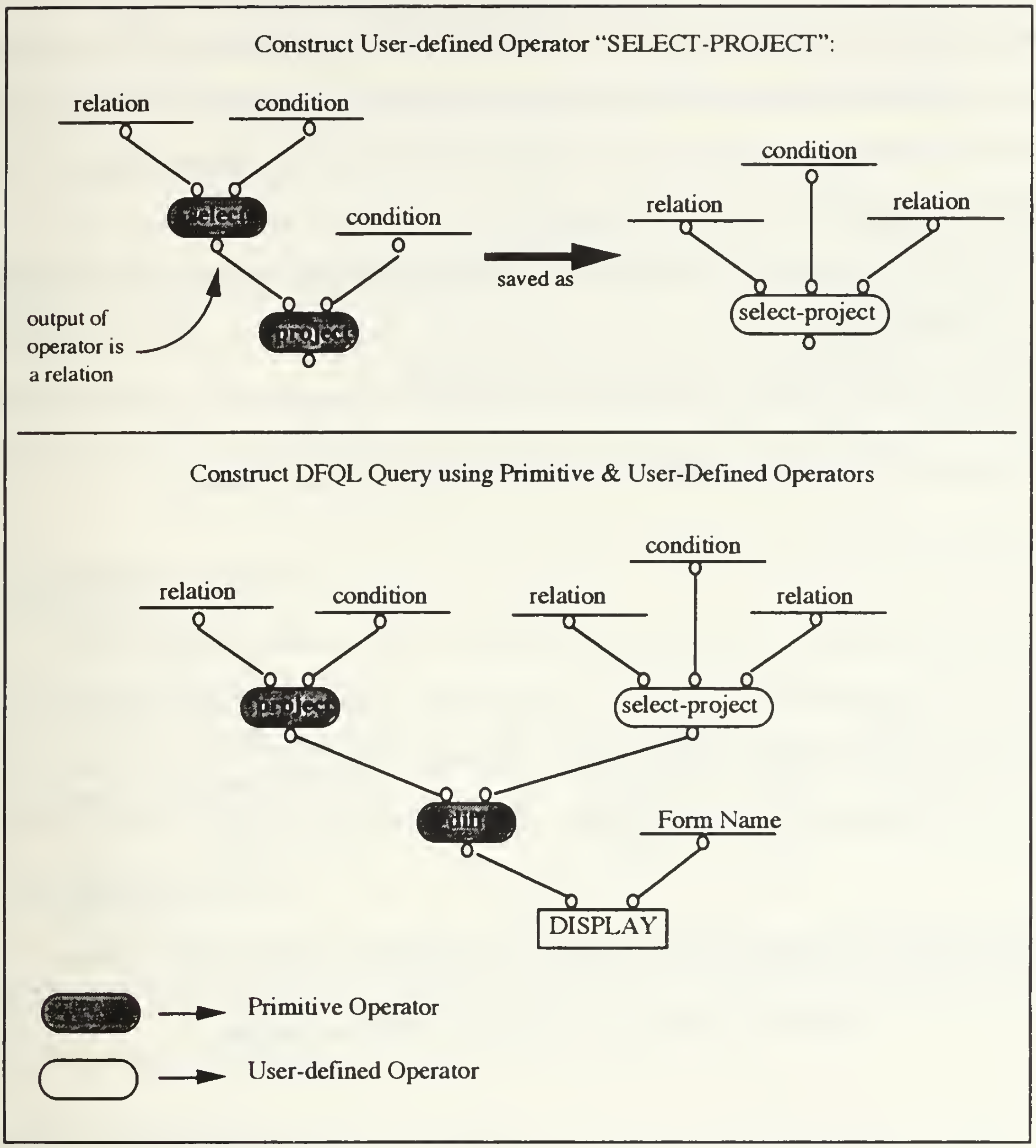


Figure 4: DFQL Query using Primitive and User-Defined Operators

G. AMADEUS

SQL has become the *de facto* industry standard query language for Relational Database Management Systems. Although ANSI and ISO standards exist for SQL, each RDBMS vendor normally supports its own SQL dialect. This presents a problem for the user of a multiple back-end[2] RDBMS, since SQL queries for one RDBMS may not necessarily run on a second RDBMS. A single front-end system that can act as an interface between users and assorted back-end RDBMS is required. *Amadeus* is a prototype for such a system, which takes an object-oriented approach for federating relational databases. The objectives of Amadeus are [Wu91 pages 8-9]:

1. Provide an easy to use, yet powerful common language for accessing different types of RDBMS, and
2. Shield the complexity of the underlying RDBMS.

Figure 5 compares the traditional DBMS arrangement and Amadeus.

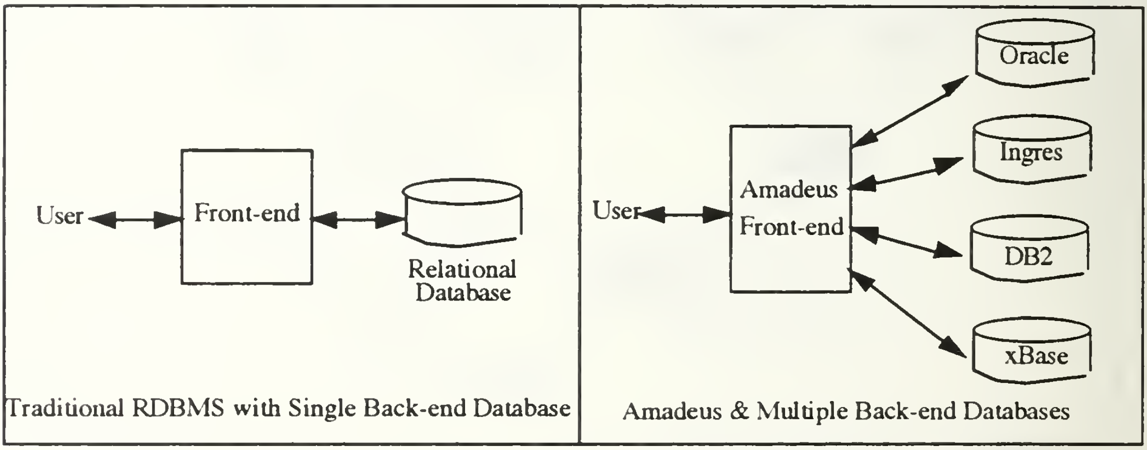

User
Front-end
Relational Database
Traditional RDBMS with Single Back-end Database
User
Amadeus Front-end
Oracle
Ingres
DB2
xBase
Amadeus & Multiple Back-end Databases

Figure 5: Comparison of Traditional DBMS and Amadeus

The key features of Amadeus are its use of the DFQL high-level visual query language and an object-oriented architecture. By using DFQL as the front-end query language, users do not have to learn different dialects for each RDBMS connected to the system. Rather, DFQL provides a consistent, easy-to-use visual front-end interface for each back-end RDBMS. Additionally, the object-oriented design of Amadeus ensures an easy to modify,

[2] Typically, a DBMS is separated into two distinct parts: the "front-end" or user interface, and the "back-end" which comprises the actual database. A multiple back-end system allows users to connect to a number of different DBMS' from a single front-end.

extensible system, which is demonstrated by the designs of both the Interface Editor module and Form Use application. Amadeus is composed of a number of modules which can be tailored to support individual user requirements. The six modules include:

1. Database Engine

This module performs all database operations by either carrying out the operations internally (by using the kernel database engine) or by delegating the operations to a back-end RDBMS. In the latter case, generated SQL statements are passed to the back-end for processing. Each supported RDBMS has its own corresponding Amadeus database engine module (e.g., Oracle database engine, Ingres database engine, DB2 database engine, etc).

2. Interface Editor

The Interface Editor module is used to design input forms, output screen displays and hardcopy reports. The Interface Editor allows or disallows certain types of control objects according to the types supported by the connected RDBMS. The design and implementation of the Interface Editor module comprises a major portion of this thesis.

3. Database Editor

The Database Editor module is used to define and modify databases. This module adjusts itself to allow different access controls to each database, depending on the connected back-end RDBMS.

4. Relation Editor

The Relation Editor module is used for defining and modifying relations. The module allows users to define relations according to the rules of the connected back-end RDBMS.

5. Query Editor

The Query Editor module is used to perform all aspects of database operations. The module allows identical DFQL diagrams to be used for querying different connected back-end RDBMS. Since each RDBMS employs its own dialect of SQL, the Query Editor generates appropriate SQL statements tailored to the connected RDBMS. All of this is transparent to the user who is formulating the query. Forms created by the Interface Editor module are used by the Query Editor to support data input and output (display). Objects required for data input, display and type checking have been developed and implemented as part of this thesis, and will ultimately be integrated into the Query Editor.

6. Program Editor

The Program Editor module is used for creating database application programs. The module is not yet implemented.

III. HUMAN-COMPUTER INTERFACE DESIGN ISSUES

A. INTRODUCTION

This chapter provides a brief discussion of Human-Computer Interface (HCI) design issues relevant to the design and implementation of the Interface Editor module and Form Use application program. Additional HCI issues are discussed in Appendix D.

Human-Computer Interface (HCI) design is a constantly evolving, multi-disciplinary field of study that focuses on computer systems and the way in which people interact with them. Contributions from the fields of computer science, human factors, psychology, graphic arts and education, play an essential role in the HCI design process. The field is relatively new[1], and is being influenced "by the tide of development - by the persistent flood of hardware and software products in the marketplace, and by the changing nature of how they are created, purchased and put into use." [Winograd90, p. 443]

1. A Brief History

Grudin identifies evolutionary stages of user interfaces by describing the following five levels of user interface *focus*. These levels correspond to general characteristics of user interfaces at various evolutionary stages, from the 1950's (level one) into the future (stages four and five) [Grudin90 p. 261-265]:

1. The interface as hardware.
2. The interface as software
3. The interface as terminal.
4. The interface as dialogue.
5. The interface as work setting.

[1] The Association of Computing Machinery (ACM) held its first conference on Computer and Human Interaction (CHI) in 1981. The ACM special interest group on Computer-Human Interaction (SIGCHI) was founded shortly thereafter, and held its first annual conference in 1983.

The first user interfaces were developed in the 1950's, and were tied directly to computer hardware. This did not present a problem, since engineers were the primary users of computers and were quite comfortable working at the machine-level. In the 1960's and 1970's, user interfaces began taking on forms which better assisted computer programmers. These new interfaces included high-level programming languages, operating systems, compilers, debuggers and assemblers. During this period (level two), the standard teletype was the preferred means of communicating with a computer. Bit-mapped graphics and cathode-ray tubes (CRTs) gradually replaced the more primitive interface devices, but for the most part, the governing concept of the time remained "the 'friendliest' user interface was the briefest user interface" [Ambler89 p. 19].

The proliferation of the personal computer in the mid-1980's saw the emergence of new computer markets aimed at the non-programmer. This was accompanied by a shift of focus in user interfaces towards visual displays and interactive computing systems (level three). This focus is intact today; is dominated by research into basic perceptual and cognitive processing issues [Grudin90 p. 264], and influences the design of the interfaces developed for this thesis.

The last two levels of user interface focus (levels four and five) involve more abstract concepts and relationships. At these levels interface actors, agents, dialogs and Virtual Reality/Virtual Environments enter into the realm of user interface design. Specifically, level four involves a higher-level cognitive focus and attempts to model end-user goals and plans, develop a sense of dialogue with the user, and create adaptive user interfaces. Level Five is directed towards the work setting, where computers are expected to play an important role in supporting group working environments. Examples of such "groupware" systems include electronic mail, co-authorship, distributed project management and group decision support. [Grudin90 p. 264-265]

The five levels of interface focus are summarized in the following table [Grudin90 p. 265]:

Table 1: SUMMARY OF THE DISTINCTIONS ACROSS LEVELS OF INTERFACE FOCUS

	Level 1 Interface as hardware	Level 2 Interface as software	Level 3 Interface as terminal	Level 4 Interface as dialogue	Level 5 Interface as work setting
Principal users	Engineers/ programmers	Programmers	End users	End users	Groups of End users
Interface specialist disciplines	Electrical engineering	Computer science	Human factors cognitive psych graphic design	Cognitive psychology, cognitive science, (dramatic arts?)	Social psych., anthropology, organizational, etc.
Research Methods	Largely informal	Largely informal	Laboratory experiment	Wizard of Oz, thinking aloud, data capture	Ethnographic, contextual, participant observer
Duration of basic events studied	Microseconds/ hours	Milliseconds/ hours	Seconds	Minutes	Days
Cost of evaluation	Lowest	Low	Moderate	High	Highest
Precision, generality	Highest	High	Moderate	Low	Lowest
Major focus	1950's	1960's-1970's	1970's-1990's	1980's-	1990's-

B. TYPES OF USER INTERFACES

1. Command Languages and the Command Line Interface

Command languages originated with operating system commands, and can be distinguished by their immediacy and by their impact on devices or information [Schneiderman92 p 144]. Commands are generally brief, transitory and produce an immediate result on some object of interest. Command languages can be extended somewhat through the use of *macros* which allow constructing reusable sequences of commands. Command languages may consist of single commands or have complex syntax. Operations may number into the thousands.

The *command-line* interface was the dominant form of user interface until the early-1980's. Despite its recognized problems (including the potential for increasing cognitive load by requiring a user to memorize a potentially large set of (not necessarily logical) commands, flags, formats and associated syntax) the command-line interface survives today in many popular systems[2]. One reason for this is simplicity. The command-line interface is not as dependent upon high-speed computer architecture as are Graphical User Interfaces (GUI). Additionally, command-line interfaces are relatively easy to program. However, this does not necessarily translate into an intuitive, easy to use interface, as evidenced by the following Unix command which blanks lines from a file:

```
GREP -V ^$FILEA > FILEB
```

Schneiderman provides the following summary of command languages in [Schneiderman92 pp. 174-175]:

> Command languages can be attractive when frequent use of a system is anticipated, users are knowledgeable about the task domain and computer concepts, screen space is at a premium, response time and display rates are slow, and numerous functions that can be combined in many ways are supported. Users have to learn the semantics and syntax, but they can initiate rather than respond, rapidly specifying actions involving several objects and options.

While command languages are appropriate for certain types of interfaces, they are inappropriate for the implementation of the Interface Editor module and Form Use application. The desired interface must be able to represent real-world objects (Forms) in a graphical environment, and allow manipulation of these objects in much the same way as a real-world Form object is manipulated. This requires a *Graphical User Interface*.

[2] MS-DOS and Unix are examples of command-line based interfaces which still enjoy wide popularity today.

2. Graphical User Interfaces

In the command-line interface, the user was restricted to working only with textual information. The evolution of hardware (and accompanying software) technology enabled researchers to move beyond the confines of the text-based interface by translating familiar every-day objects into the Human-Computer Interface. For the first time, users could manipulate information on a computer monitor in much the same way as they did in the real-world. The computer screen became a metaphor for a *desktop* which contained windows, icons, menus and other graphical objects. The concept of What-You-See-Is-What-You-Get (WYSIWYG) was introduced in word-processing, so that the representation of a page on a computer screen was identical, in nearly every respect, to the printed output. Multiple windows could be overlaid onto the desktop. Each window could be a terminal (or shell) on the computer, a terminal onto another machine, or the interface to a software application (such as word-processing/desktop publishing, database, spreadsheet, or graphics design packages). [Schneiderman92 page 198; Locke page 11]

There are two major consequences (from a user's perspective) of the graphical user interface. The first is that the user is more isolated from the operating system, and is better able to concentrate directly on performing a task rather than having to interact with the operating system in order to perform the task. For example, to open a file in a graphical interface a user might select the document by clicking on its icon with a mouse key. The underlying operating system commands for opening the document are invoked *by the interface*. In a command-line interface, operating system commands to open the document are invoked explicitly *by the user*.

Secondly, the graphical interface lead to the concept of *consistency*, whereby certain properties of an application program's user interface possess predictable characteristics and behaviors. Thus, a consistent interface is one in which specific commands always result in the same action(s) and produce the same result(s). The Interface Editor module presents a consistent interface for designing and editing Form objects.

3. Direct Manipulation

Direct manipulation refers to a type of graphical user interface in which the user operates on a representation of the objects(s) of interest [Schneiderman92 p. 33]. Typically, some type of a pointing device (such as a mouse, track-ball or pen and tablet) are employed to permit user interaction with objects which appear on the computer screen. The Interface Editor module employs a direct manipulation interface, which is discussed in Chapter VI.

C. EVALUATION AND USEABILITY OF USER INTERFACES

This section presents evaluation guidelines and usability parameters which have been incorporated into the design of the Interface Editor module and Form Use application. Usability parameters tend to reflect a user's attitude towards a specific interface, while evaluation guidelines outline key principles for good HCI design.

1. Evaluation

Jakob Nielsen proposes the following four methods for evaluating a user interface [Nielsen90a]: *formally, automatically, empirically* and *heuristically*. Of these methods, the heuristic method has been applied to the design of the interfaces developed for this thesis.

Heuristic evaluation essentially entails looking at an interface and deciding what is good and what is bad about it. To be consistent the evaluation should be based on a set of established guidelines which, in practice, can easily number in the hundreds or thousands. This is especially true if the guidelines define a specific interface standard. In order to make the heuristic evaluation process more manageable, Nielsen and Molich propose using the following set of heuristics which were chosen for their ability to explain a large proportion of the problems encountered in interface designs [Nielsen90a].

1. Simple and natural dialogue
2. Speak the user's language
3. Minimize user memory load[3]
4. Be consistent

5. Provide feedback
6. Provide clearly marked exits
7. Provide shortcuts
8. Good error messages
9. Prevent errors

2. Usability

Nielsen discusses five generally accepted *usability parameters* for computer systems while applying them specifically to hypertext[4] systems. [Nielsen 90b] These five parameters are:

1. **Easy to learn**. A new user can quickly get some work done with the system.
2. **Efficient to use**. Once a user becomes familiar with the system, a high level of productivity is possible.
3. **Easy to remember**. After an absence from the system, the average user can quickly get back "up to speed" on the system with little effort or re-training required.
4. **Few errors**. Use of the system does not in itself promote errors. A user can easily recover from an error if/when one occurs and the system must be immune to catastrophic errors.
5. **Pleasant to use**. Users like using the system (or, possibly more common, users do not dread using the system).

D. DIALOG DESIGN

Dialog is essential to the successful interface design for interactive systems. Dialog encompasses everything related to a user's interaction with a system, including user input

[3] Studies of human information processing indicate that human channel capacity or processing power is limited to roughly 2.5 bits of information, which translates to about seven (plus or minus two) items.

[4] Hypertext refers to a text system consisting of non-sequential text nodes and connecting links.

and command selection, system feedback (such as alert, error and status messages) and system error handling.

1. Dialog Design Rules

Schneiderman provides the following *Eight Golden Rules of Dialog Design*: [Schneiderman92 pages 72-74] which have been incorporated, to varying degrees, into the design of the user interface developed for this thesis.

1. **Strive for consistency**. Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus and help screens; consistent commands should be employed throughout. The interface which controls Form design and editing in the Interface Editor module are essentially identical, presenting a consistent interface. Help windows are identical, although the topics for each window differ depending on the window (Design Form, Edit Form, Tab Order, Form View) which the user is currently working in.
2. **Enable frequent users to use shortcuts**. This is supported in the Interface editor by the inclusion of *command keys* and an *icon bar* (discussed in chapter IV).
3. **Offer informative feedback**. For every operator action, there should be some system feedback. Each action in the Interface Editor and Form Use applications provides some form of user feedback. Icon Buttons invert when selected, dialogs sound an alert when activated, the name of the active Form (in the Form View window) changes as a Form is opened or closed.
4. **Design dialogs to yield closure**. Sequences of actions should be organized into groups with a beginning, middle, and end. This principle is incorporated into the command sequence required to design and edit a Form object.
5. **Offer simple error handling**. Design the system so the user can't make a serious error. The system should be able to detect errors and offer simple, comprehensible mechanisms for handling the error.

6. **Permit easy reversal of actions**. As much as possible, actions should be reversible. This is the most difficult rule to implement. Reversal of actions is permitted only to a limited extent in the Interface Editor module. Users have the option of saving or discarding (without saving) Forms from the Design Form window. While editing a Form, changes can be discarded without saving (essentially reversing a sequence of editing commands). Certain actions, such as deleting a Form from disk, can not be reversed.
7. **Support internal locus of control**. Users want to feel in control. Surprising system actions, tedious sequences of data entries, incapacity or difficulty in obtaining necessary information, and the inability to produce the action desired all build anxiety and dissatisfaction.
8. **Reduce short-term memory load**. Keep displays simple. Offer on-line help and assistance to the user. A simple on-line help system has been implemented in the Interface Editor and Form Use applications. Additionally, program control features have been kept to a minimum without sacrificing the power and flexibility of the interface.

2. User Feedback

System feedback should be concise, descriptive and informative. A user should not have to question the meaning of a system-generated error or status message. Interfaces which isolate a user from the underlying operating system (such as the Macintosh Finder) have the potential to confuse a user with poorly phrased dialogs which assume (or require) intimate knowledge of the operating system. The user feedback features of the Interface Editor and Form Use applications have been designed to avoid this type of problem. In addition to the eight rules of dialog design listed in the last section, system feedback to the user (including alerts, dialogs, prompts and error messages) for these two interfaces also includes the following guidelines [Apple85]:

1. Use plain language.
2. Use an active voice.
3. Phrase messages so that they are unambiguous.
4. Use icons whenever possible. Graphics can describe some errors better than words, and familiar icons can help users better distinguish their alternatives.
5. Dialogs should be informative, providing enough information to enable the user to take the appropriate action.
6. Never refer the user to external documentation for further clarification.

IV. DESIGN AND IMPLEMENTATION

A. DEVELOPMENT PROCESS

The Prograph programming environment lends itself extremely well to both structured and evolutionary development processes [TGS90b, pages 230-231]. Structured development is normally associated with a "top-down" design strategy employing a structured, analytic approach to system design. Program code is not written until the project has been completely specified. All documentation associated with the system (e.g., requirements, design and test documents as well as the actual program code), is rigorously maintained. Any changes to the system are implemented in a manner similar to the original design effort, and can be considered a mini-development process.

Evolutionary development, on the other hand, is usually associated with creative prototyping styles such as those found in research settings where the goal is to explore new ideas without being bound by rigid documentation and specification rules. Thus, changes can be made quickly and easily.

Prograph's seamless editor/interpreter environment provides the tools necessary to create and edit applications "on the fly", easily accommodating the evolutionary approach to systems development. For this reason, and the fact that interface design in general requires a great deal of flexibility while trying out new ideas, an evolutionary design approach was chosen for this thesis. The visual nature of Prograph makes an application both a system model and an executable program. In this respect, the application code provides up-to-date dataflow diagrams of the Interface Editor module at each stage of development.

B. USER INTERFACE CONSIDERATIONS

Two user interfaces were developed for this thesis: the *Interface Editor* module interface and the *Form Use* application program interface. A central feature of the Interface Editor's user interface is the incorporation of the following four separate methods for controlling program action:

1. A menu bar (with associated pull-down menu commands).
2. Window Buttons.
3. Command-key equivalents to menu commands.
4. An Icon Bar.

The first three methods are included to conform to existing Macintosh user interface guidelines. Most casual users of graphical user interfaces are familiar with button objects and the menu bar, while more experienced users are generally comfortable with command key equivalents for menu bar items. The fourth method is the *icon bar*, which is located at the top right-hand corner of the Interface Editor's main window. The decision to include an icon bar was based on a desire to extend the traditional Macintosh interface, provide an alternative means of program control for the user and to explore icon development issues in general. Icon buttons are also included as the control mechanism for the data input and display windows developed for the Form Use application, and provide a consistent control interface for all Amadeus-related Form manipulation actions.

Icon design presents a number of difficulties, chief among them the fact that what may be clear to one person may be completely obscure to another. Alan Kay describes this problem as a consequence of *semantic focus*, having to do with the amount of meaning and connectivity that can be solved by looking at a diagram [Kay in Laurel91, p. 202].

Poorly-designed icons do not readily suggest the underlying metaphor or associated program action. In such cases, a user is forced to learn what the icon actually does rather than what it suggests. This can be a difficult undertaking given an interface with dozens of icons, a popular practice in a growing number of commercial software products today.

C. THE INTERFACE EDITOR

1. Program Control Decisions

Menu bar commands (and their command-key equivalents) remain hidden from view until the appropriate menu item is selected, at which time its associated menu items become visible. The icon bar, however, remains visible whenever the main module window is active. Users are generally more comfortable in an environment where objects are brought to them instead of having to search for the objects. This is referred to as *user-centered design* [Tognazzini91 p. 172]. The inclusion of the icon bar satisfies this user-centered design criteria, eliminating the requirement for a user to constantly search pull-down menus or remember command-key combinations to select specific program control actions.

Commands relating to Form management (New, Open, Close, Save, Save As, Print) are found in both the icon bar and the *Forms* menu bar menu. Similar commands are found in other Amadeus modules in a *Database* menu (for database management). The *Forms* and *Database* menu contents are similar to the *File* menu commands found in the Macintosh Finder. This presents a consistent interface across the Amadeus application and Finder. Thus, there should be no confusion, for example, as to the meaning of the *Open* command (which opens a *Form* in the Interface Editor Module, a *Database* in the Amadeus database definition module, and a *File* in the Macintosh Finder). This is also consistent with general object-oriented programming concepts, since a single command (message), in this case *Open*, results in different actions depending on the receiving object (*Forms* menu, *Database* menu or *File* menu).

Not all Interface Editor icon bar commands are available as menu bar selections. Commands without parallels in Amadeus' *Database* and the Macintosh Finder's *File* menus are found only in the icon bar, and include Edit Form, Delete Form, Help and user Preferences controls. The decision not to include these commands in the *Forms* menu contributes to the overall consistency of the user interface.

2. Icon Design

Icons included in the Interface Editor present images commonly found on the Macintosh desktop. Where no desktop correlations could be found, text was incorporated as much as possible in an icon's graphic design. The intent of including icons in the user interface was not to introduce a completely new set of metaphors, but to build on what a user is (theoretically) already familiar with. The choice of individual icons was made after examining commercial software products which employ icons for program control and a survey of literature on icon design. The icon buttons themselves were given color and three-dimensional shading to make them stand out against the rest of the interface. Muted colors were chosen to avoid overpowering the user's senses.

It is not reasonable to assume that every user will correctly identify the purpose of each icon the first time the interface is used. For this reason, consideration was given to including text labels under each icon button identifying its function (e.g., "open" beneath the open icon button). However, this tended to clutter the icon bar and required the use of very small text which proved difficult to read. Ultimately a decision was made not to include identifying text for each icon button. To compensate for the lack of amplifying text, the Macintosh System 7 Balloon Help feature (which is supported by Prograph version 2.5) is used to provide a brief synopsis of each interface object (buttons, icons, menus, menu items and other window items).

Space has been left in the icon bar to accommodate additional icon buttons should the need arise. ResEdit[1] and Prograph's *Icon Button class* (discussed later in this section) provide a straightforward means of adding icon buttons to the interface, or modifying existing icons, as required.

[1] ResEdit is a resource editor utility available from Apple Computer, Inc. which allows editing, creation and deletion of a Macintosh application program's resource data.

3. End-User Views

The Interface Editor module's design assumes two general types of users: database administrators (who create databases and data entry/display Forms) and data entry/retrieval personnel (who use the Forms). This distinction led to a decision to implement the Interface Editor as a separate application rather than integrate it into the Query Editor. One consequence of this decision was the loss of a straight-forward connection to the database definition of the active database. The solution to this problem was the creation of an external database definition disk file to substitute for the ability to read information directly from an active database's class attributes. However, the decision lead to a more manageable and maintainable system, due to the smaller program size and accompanying modular design.

4. Class Descriptions

This section describes classes unique to the Interface Editor module. Prograph System Classes will not be discussed. Figure 1 shows the Interface Editor module class hierarchy.

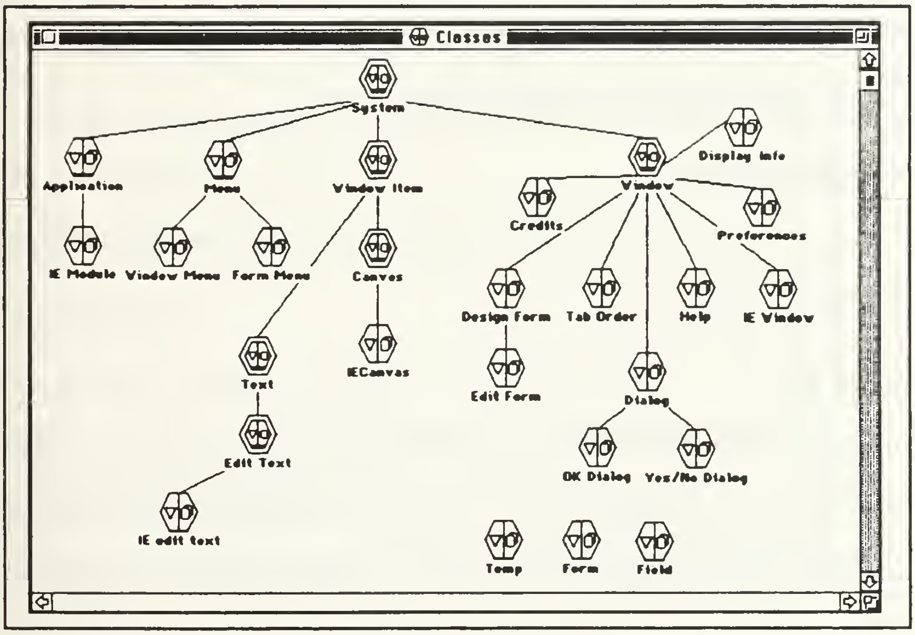


Figure 1: The Interface Editor Module Class Hierarchy

a. IE Window

This class controls the user's view of the Interface Editor module. The main module window is an instance of this class. All menu, button and icon bar selections from the main window are handled by methods defined in this class.

b. Help

Help windows are defined for the following Interface Editor windows: Interface Editor Window, Define Form, Edit Form. All help windows are instances of this class.

c. Tab Order

This class allows the user to define the tab order of a Form. The tab order of a Form is determined by the order in which a field appears in a Form's field list. Altering the order of a field in this list changes the tab order of the Form. The tab order window is an instance of this class.

d. Preferences

This class allows the user to change the foreground and background colors of a Form. Supported colors include: black, white, red, green, blue, cyan, magenta and yellow. The User Preferences window is an instance of this class.

e. Design Form

This class allows the user to define new input Forms. The Design Form window is an instance of this class.

f. Edit Form

This class allows the user to edit the Form which is displayed in the Interface Editor's main window (the currently active Form). The Edit Form window is an instance of this class. The Edit Form Class is a descendant of Design Form, and inherits its methods.

g. Credits

Each Macintosh application program has an *About* menu item in its Apple Menu. Selecting this item displays general information about the application. The About window is an instance of the *Credits* class, and displays programmer credits.

h. Display Info

This class controls display of field information. Double-clicking on a field of the active Form in the *Form View* window opens a window which lists the characteristics (attributes and attribute values) of the particular field. The Display Info window is an instance of this class.

i. Dialog

The Dialog class contains the methods for displaying dialog boxes in response to user actions.

j. OK Dialog

This class provides the window definition for a dialog which contains only one user choice (OK), which is used to alert the user to a specific program condition.

k. Yes/No Dialog

This class provides the window definition for a dialog which contains two user choices (YES and NO), and is used to obtain user confirmation before a specific program action is executed.

5. User Interface

This section describes the user interface for the Interface Editor module. Dialog design rules and guidelines for evaluation and usability of user interfaces, discussed in Chapter III, have been applied to the overall design and implementation of the user interface for both the Interface Editor and Form Use application program.

a. Menu Commands

The Interface Editor contains three separate menus in its menu bar: FILE, WINDOW and FORMS. Figure 2 shows the menu bar.

Figure 2: The Menu Bar

The FILE menu is a default menu item created by the Prograph Application Editor. It contains only one command: QUIT, which closes all active windows and terminates program execution.The Interface Editor module is activated by selecting the INTERFACE EDITOR command from the WINDOW menu. The FORMS menu enables a user to perform certain basic Form management operations. The FORMS menu includes the following commands:

1. **New.** This command opens a window titled *Design Form* which allows the user to create a new, untitled Form. The new Form is given a name the first time it is saved.
2. **Open.** This command displays a standard Macintosh Open dialog containing a scrolling list of files. Clicking the *OPEN* button or double-clicking on a File name from the scroll list will open the selected (highlighted) Form in the Interface Editor's Form View Window.
3. **Close.** This command closes the active Form. If the Form has been modified since the last time it was saved, a dialog is displayed which allows the user to save the Form, or dismiss the dialog and close the current form without saving it.
4. **Save.** This command saves the active Form to a disk file. If a file already exists with the same name as the active Form, the file is overwritten.
5. **Save As.** This command opens a standard Macintosh Save dialog which allows the user to specify a new name for the active Form. If the Form is saved under a new

name, the active Form is renamed, saved to disk (with the new name), and the old file is closed (retaining its old file name and Form definition).

6. **Page Setup.** This command allows the user to specify printing parameters such as paper size and orientation.
7. **Print.** This command prints the active window.

b. The Icon Bar

The icon bar consists of ten icon buttons which allow the user to access Interface Editor commands independent of the FORMS pull-down menu. In addition to the commands found in the FORMS menu, a number of other commands are included in the icon bar. Figure 3 shows the icon bar.

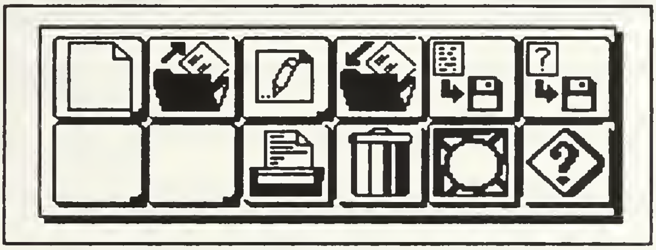


Figure 3: The Interface Editor Icon Bar

Icon buttons descriptions (from left to right, top to bottom) are:

1. **New.** This command is the same as described in the FORMS menu.
2. **Open.** This command is the same as described in the FORMS menu.
3. **Edit.** This command is the same as described in the FORMS menu.
4. **Save.** This command is the same as described in the FORMS menu.
5. **Save As.** This command is the same as described in the FORMS menu.
6. **Print.** This command is the same as described in the FORMS menu.
7. **Delete.** This command allows the user to delete the active Form from disk. The Delete command can not be undone. A dialog prompt is displayed giving the user the

opportunity to cancel the command or confirm the selection before a Form is permanently deleted.

8. **Preferences.** This command allows the user to select background and foreground colors for displaying a Form in the Interface Editor. These preferences do not become part of the Form's definition, and are used only by the Interface Editor.
9. **Help.** This command opens a user help window.

6. Windows

The Interface Editor window is the main window for the Interface Editor module, and is shown in Figure 4. In addition to the icon bar described earlier, the following items appear in the window:

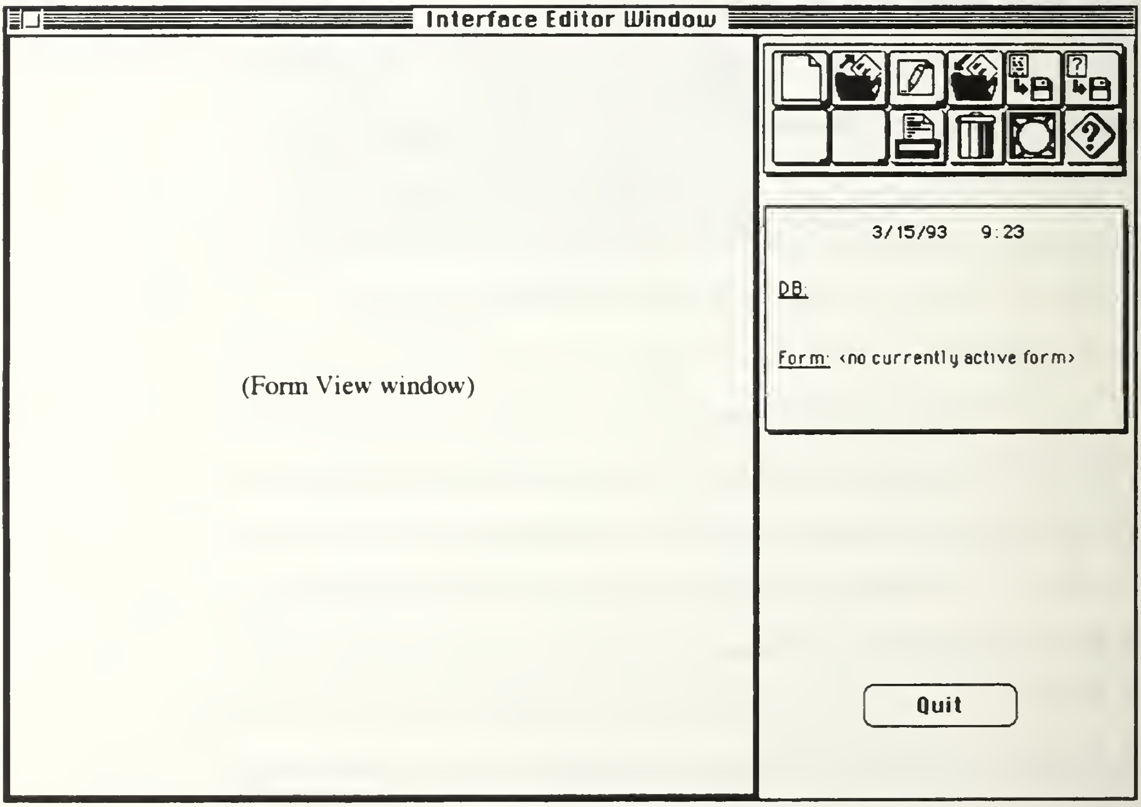


Figure 4: The Interface Editor Window

1. **Current Form Title.** This command displays the name of the active Form.
2. **Data and Time.** The current date and time are displayed at the top right of the window, just below the icon bar. The date and time are controlled by the Interface Editor Window's *Idle Method* which is defined in the Application Builder's Window editor.
3. **Active DB.** This box displays the name of the database which the Form is being designed for.
4. **Form View Window.** This window occupies about two-thirds of the Interface Editor window area, and is actually a Macintosh *canvas* object which supports Quickdraw[2] editing and graphics. When the Interface Editor module is first activated, the form view window appears black in color, signifying an empty form view window (i.e., no active or open Form). Additionally, the Form name display reads "<no currently active form>. When a Form is opened, the form view window background changes to the user-defined background color, and rectangles representing fields of the current Form appear, along with corresponding labels. When the active Form is closed or deleted, the form view window once again becomes black in color. Each field of a Form is a descendent of the Prograph *Window Item* class. In order for objects of this class to be visible in a window, they must first be included in the window's item list (a list of all window items belonging to the window). Displaying a Form in the form view window is a two-step process. The location of each Form field is determined relative to the window, and appropriate field attributes are set to reflect this data. Once field locations have been determined, each field object is added to the item list of the Interface Editor window (the *owning* window of the canvas object). This is not sufficient, however, to permit graphical manipulation of field objects. Graphical operations such as *dragging* and *resizing* of objects is accomplished in a canvas object. Since a canvas object is itself a descendent of *Window Item*, it can not contain other *Window Item* objects. The solution to this problem is to place a canvas (the form

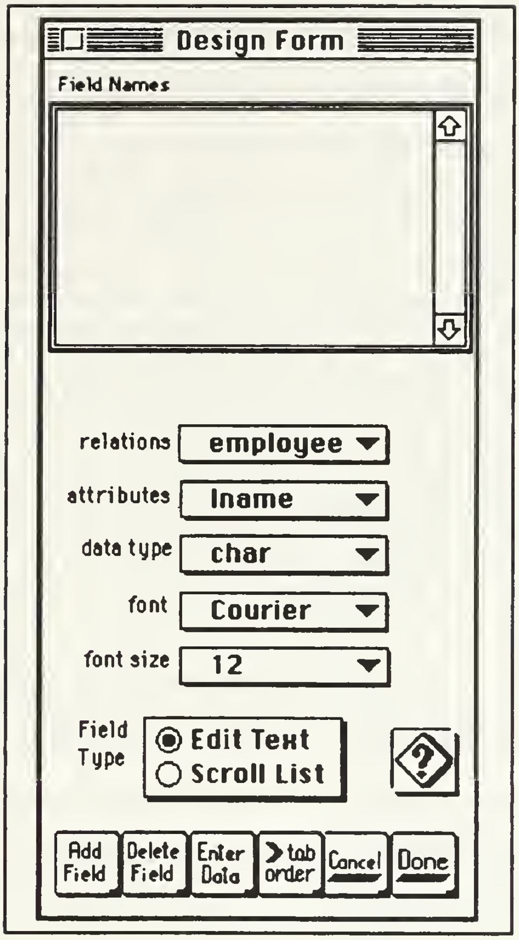
[2] Quickdraw is the part of the Macintosh Toolbox that supports creation of complex graphic operations [Apple85 p. I-137].

view window) on top of the Interface Editor window. Canvas objects (in this case, *rectangles*) are then added to the *canvas* directly on top of each field object. Thus, a Form is displayed as a 2-layer screen display. The bottom layer contains the actual field object and the top layer contains the rectangle which bounds the field objects. Whenever a field is moved or resized, only the canvas object is actually manipulated directly. The corresponding field object (in the window item list), is updated using positional information obtained from its corresponding canvas rectangle object. This layering process is not required when Forms are displayed in Input and Display Form windows by the Query Editor, since graphical operations on fields are not permitted in these windows.

5. **Quit Button.** Selecting this button quits the Interface Editor module.
6. **Close Box.** A *close box* is located in the upper left-hand corner of the window. Clicking in this box is equivalent to selecting the Close command in the FORMS menu or clicking the QUIT button.

a. The Design Form Window

The Design Form window is activated when a user selects the *New* menu item, types a command-N sequence from the keyboard, or selects the *New Form* icon from the icon bar. Figure 5 shows the Design Form window. This window allows a user to create a new form object, and consists of the following objects:



relations: employee

attributes: lname

data type: char

font: Courier

font size: 12

Field Type: ● Edit Text ○ Scroll List

Figure 5: The Design Form Window

1. **Field Names.** This is a scroll list which contains the names of the fields defined for the form being designed.
2. **Name.** This is an edit text object which is located directly below the Field Names scroll list and allows a user to enter field names from the keyboard. If a field name is too long to fit into the viewable portion of the edit text object, the text automatically scrolls to the right as the user types. Each field name must be unique.
3. **Field Type.** This is a radio set object which allows a user to select the object type of the field. The values of the radio set are static, independent on any database definition and can not be changed by the user.

4. **Relations.** This is a pop-up menu object which contains the names of the relations of the database which the Form is being designed for.

5. **Attributes.** This is a pop-up menu object which contains the names of the attributes associated with the relation which is currently visible in the relations pop-up menu.

6. **Data Type.** This is a pop-up menu object which contains the name of the data type associated with the attribute which is currently visible in the attributes pop-up menu.

7. **Font.** This is a pop-up menu object which contains font names. The font name effects the appearance of data entered into the associated Form field.

8. **Font Size.** This is a pop-up menu object which contains font sizes. The font size effects the appearance of data entered into the associated Form field.

9. **Add New Field Button.** Selecting this button activates the Name field to accept text from the keyboard.

10. **Enter Data Button.** Selecting this button enters the description of the field whose name is currently highlighted in the *Field Names* scroll list. A field's description is defined by the text which appears in the Name field, the values which appear in the windows pop-up menus, and the selected value of the Field Type radio button set.

11. **Delete Field Button.** Selecting this button deletes the field whose name is currently highlighted in the *Field Names* scroll list from the Form.

12. **Tab Order Button.** Selecting this button activates a separate modal window (the *Tab Order Window*) which allows the user to define the order in which fields are accessed when the Tab key is selected.

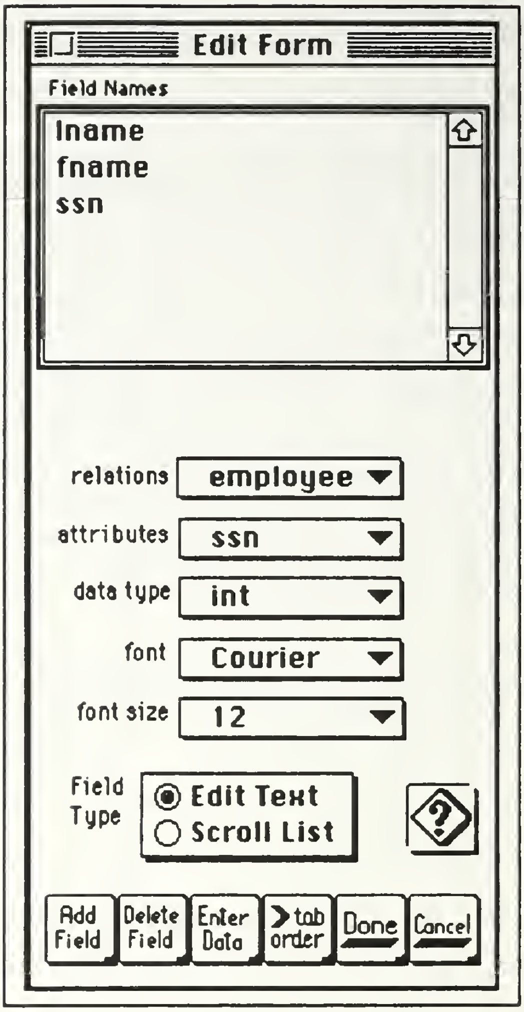
13. **Cancel Button.** Selecting this button cancels all changes made to the Form and closes the *Design Form* window without saving the Form.

14. **Done Button.** Selecting this button activates a dialog box which prompts the user to name the Form which is being designed. The user has the option of naming and saving the Form, cancelling the dialog and returning to the *Design Form* window, or closing the *Design Form* window without saving the Form.

15. **Help Button.** Selecting this button activates a help window associated with the *Design Form* window.

b. Edit Form Window

The *Edit Form* window is similar in appearance and function to the *Design Form* window with transparent differences in the underlying implementation of the controlling methods and the window title which appears at the top of the window. The *Edit Form* window can only be accessed if a Form is currently active in the *Form View* window, and allows editing of the active Form object. When the window is opened, it contains a list of the active Form's fields (in the currently defined tab order). Selecting a field name in the Field Names scroll list causes the values of the field (*Name, Field Type, Relations, Attributes, Data Type, Font* and *Font Size*) to be displayed in their respective edit text box, radio set and pop-up menus. *Edit Form* control buttons are identical to those described for the *Design Form* window. Figure 6 shows the *Edit Form* window.



Field Names

lname
fname
ssn

relations: employee

attributes: ssn

data type: int

font: Courier

font size: 12

Field Type: ● Edit Text ○ Scroll List

Figure 6: The Edit Form Window

c. *Tab Order Window*

The *Tab Order* window displays the names of the fields of a Form in the currently defined tab order. The purpose of the window is to allow the user to modify the tab order of a Form. Figure 7 shows the Tab Order window. The window contains the following objects:

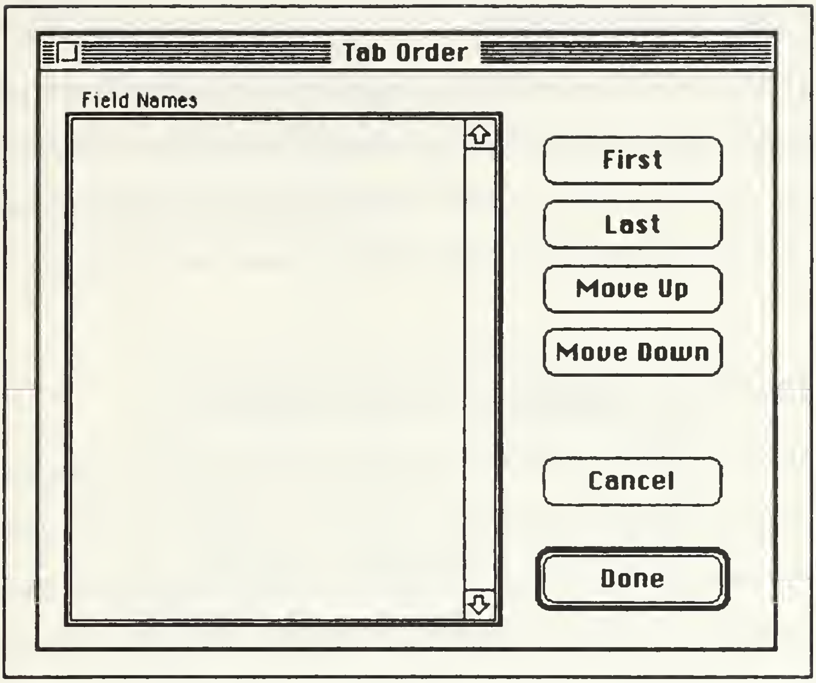


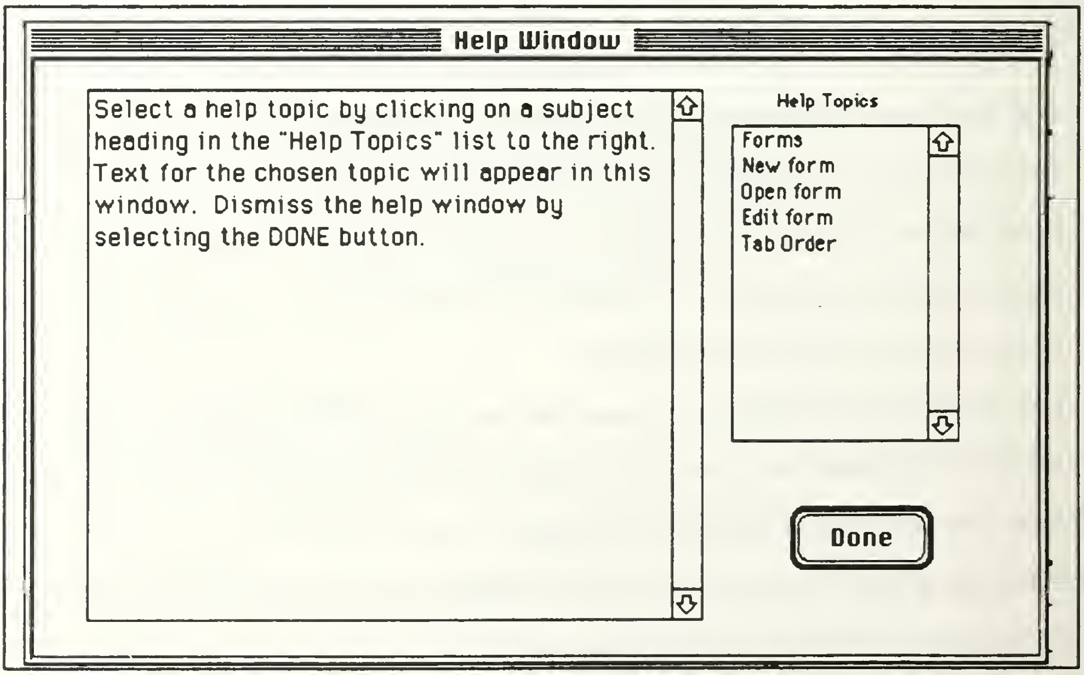
Figure 7: The Tab Order Window

1. **Field Names.** This is a scroll list identical to the *Field Names* scroll list contained in both the *Design Form* and *Edit Form* windows. Selecting a name in this list causes the field associated with the name to be the object of the window's manipulation buttons.
2. **First Button.** Selecting this button causes the field which is highlighted in the *Field Names* scroll list to be moved to the top of the scroll list (i.e., it becomes the first field in the tab order defined for the Form).
3. **Last Button.** Selecting this button causes the field which is highlighted in the *Field Names* scroll list to be moved to the bottom of the scroll list (i.e., it becomes the last field in the tab order defined for the Form).
4. **Move Up Button.** Selecting this button causes the field which is highlighted in the *Field Names* scroll list to be moved up one position in the scroll list.
5. **Move Down Button.** Selecting this button causes the field which is highlighted in the *Field Names* scroll list to be moved down one position in the scroll list.

6. **Cancel Button.** Selecting this button cancels all tab order changes made to the Form and closes the *Tab Order* window.
7. **Done Button.** Selecting this button closes the *Tab Order* window and displays the field names in the new tab order within the *Design Form* or *Edit Form* window (depending on the window which was active when the Tab Order window was opened). The new tab order will not actually be saved until the Form is subsequently saved.

d. The Help Window

Help windows provide on-line user help relating to the currently active window. Help topics are selected from the list of Help Topics by double-clicking on a topic title. Text for the selected topic appears in a scroll window. The Help window is dismissed by selecting the Done button. Figure 8 shows the Help window for the Interface Editor main module window.

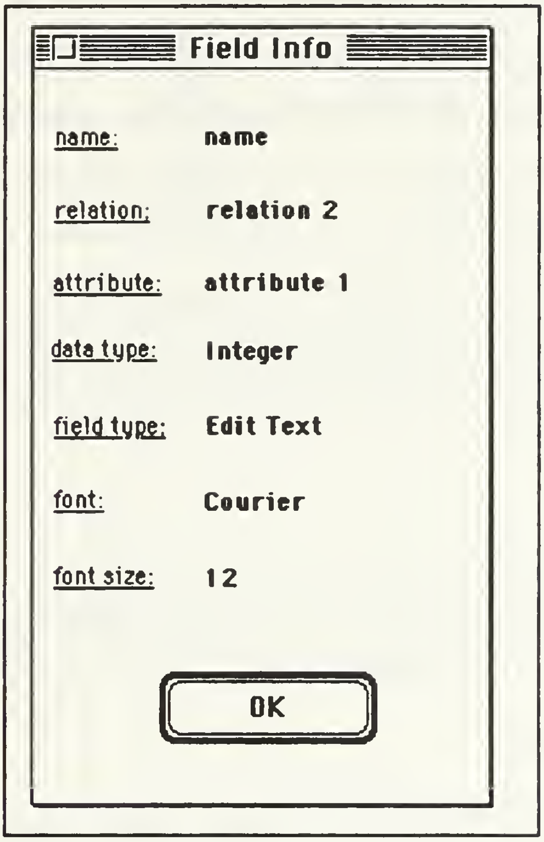


Select a help topic by clicking on a subject heading in the "Help Topics" list to the right. Text for the chosen topic will appear in this window. Dismiss the help window by selecting the DONE button.

Figure 8: The Help Window

e. *Field Information Window*

The *Field Information* window is activated by double-clicking on a field of the active Form (the Form currently displayed in the *Form View* window), and displays information about the selected field. The window is dismissed by selecting the OK button. Figure 9 shows the Field Information window.



Field Info

name: name

relation: relation 2

attribute: attribute 1

data type: Integer

field type: Edit Text

font: Courier

font size: 12

Figure 9: The Field Information Window

D. THE FORM USE APPLICATION

The Form Use application program simulates the user interface of the Query Editor module and the module's interaction with a disk-resident Form. Objects defined for this application will form the basis of the Query Editor's Form display, data entry and data validation (type checking) functions.

1. Program Control Decisions

Since this application will ultimately be incorporated into the Query Editor module, menu bar commands have been kept to a minimum. The *Input Form* and *Display Form* menus allow a user to open either data entry or data display windows. Additionally, users must retrieve Form objects from disk through the standard Macintosh Open dialog. Window activation and Form retrieval will become transparent to system users once the Form Use functions have been integrated into the Query Editor.

In order to provide consistency for Form creation, editing and use, icon buttons were included in the Input Form and Display Form windows to control data entry and display functions.

2. Class Descriptions

This section describes classes unique to the Form Use application. Prograph System Classes will not be discussed. Figure 10 shows the application's class hierarchy.

a. Form Window

This class defines the basic window in which Forms are displayed.

b. Input Form Window

This window inherits the basic window properties from the *Form Window* class, and adds control objects and methods for inputting data into a Form.

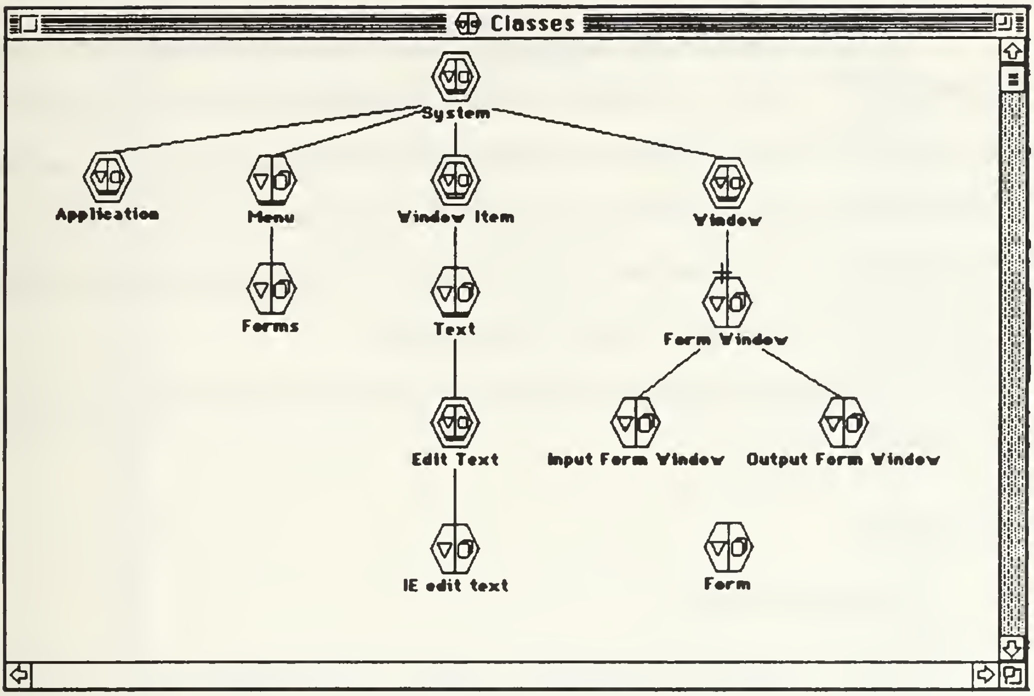


Figure 10: The Form Use Application Program Class Hierarchy

c. *Display Form Window*

This window inherits the basic window properties from the *Form Window* class, and adds control objects and methods for displaying data from a DFQL query in a Form.

d. *Form*

This class has been imported directly from the Interface Editor module, and contains the definition for a Form object.

e. *IE Edit Text*

This class has been imported directly from the Interface Editor module, and contains the definition for IE edit text objects. Methods have been added which support data input and display via IE edit text objects.

3. User Interface

The menu bar contains two items: FILE and FORM. The FILE menu contains only one item: QUIT. This is a standard default Prograph menu item, and allows the application user to terminate program execution. The FORM menu contains two items: INPUT FORM and OUTPUT FORM. The INPUT FORM command displays a Form (specified by the user) in an *Input Form* window. The OUTPUT FORM command displays a Form (specified by the user) in a *Display Form* window.

The Form Use user interface is limited to a set of icon control buttons in the Input Form and Display Form windows.

4. Windows

a. Input Form Window

Figure 11 shows the Input Form window. Forms created by the Interface Editor module are displayed in this window for data entry purposes. The window contains the following control buttons:

1. **Clear All**. This button clears all data entered into the displayed Form.
2. **Clear Form**. This button clears all data currently visible in the displayed Form.
3. **Enter Data**. This button adds the data currently visible in the displayed Form into the attribute *result* of class *Form Window*.
4. **Done**. This button is essentially the same as the Enter Data button, except that the Input Form window is closed after the displayed data has been added to the list.
5. **Cancel**. this button cancels the data entry process, deletes all data currently contained in the attribute *result* of class *Form Window* and closes the Input Form window.

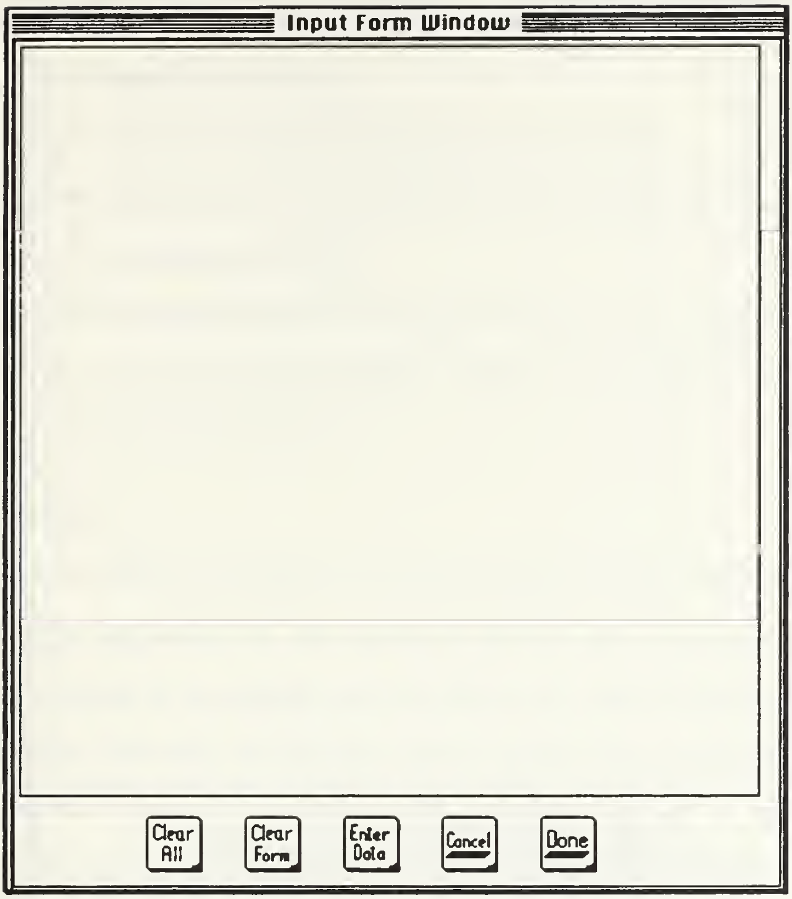


Figure 11: Input Form Window

b. Display Form Window

Figure 12 shows the Display Form Window. Forms created by the Interface Editor module are displayed in this window for data display purposes. The window contains the following control buttons (from left to right):

1. **First**. This button displays the first element of a data list.
2. **Last**. This button displays the last element of a data list.
3. **Previous**. This button displays the previous element of a data list.
4. **Next**. This button displays the next element of a data list.
5. **Done**. This button closes the Display Form window.

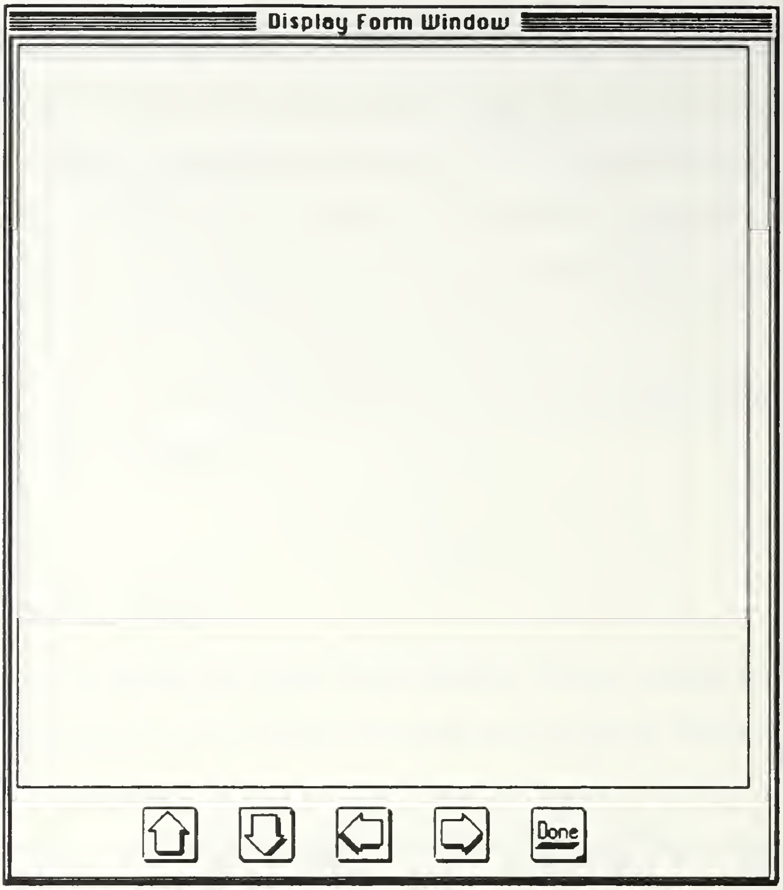


Figure 12: Display Form Window

E. THE FORM OBJECT

1. Form Object

A *Form* is an object composed of one or more *fields*, and defines a template for entering and displaying data in response to DFQL queries specified in the Amadeus Query Editor module. The Interface Editor module provides a means of defining, modifying and saving Form objects.

A Form object consists of the following attributes:

1. **name**. The Form's name is assigned by the user in the Interface Editor module. The *name* attribute is used as an input to a DFQL query, and identifies which Form is to be opened in response to the query.
2. **fields**. This is a list of *field* objects associated with the Form.
3. **end user**. Identifies the owner of the Form (i.e., the name or identifier of the user or class of user who created the Form).
4. **protected?** This attributed identifies whether the Form is protected (e.g., read-only, read-write, etc.). User access and security issues are being addressed separately, and are beyond the scope of this thesis.

2. Field Object

Each Form *field* is a Prograph Window Item object (a descendent of the Window Item class). Fields supported by the Interface Editor are considered *display fields* [Gibson90], and consist of an editable area on the screen used for editing, displaying, or updating a database. Currently, the Interface Editor creates fields consisting of descendents of the *Edit Text* Window Item class (called *IE edit text*). Extending the module to allow additional types of fields requires defining new subclasses within the Interface Editor class hierarchy (subclasses of *Window Item*) for each new field type. Each field object consists of the following attributes:

1. **name**. This value identifies the field with a unique character string. The field name is assigned by the user in the Interface Editor module at the time the field is defined.
2. **relation**. This value identifies the database relation which the field is associated with. This value is assigned by the user in the Interface Editor module at the time the field is defined.
3. **attribute**. This value identifies the database attribute which the field is associated with. This value is assigned by the user in the Interface Editor module at the time the field is defined.

4. **font**. This value identifies the font name of the field. Data displayed in the field will appear in this font.
5. **font size**. This value identifies the font size of the field. Data displayed in the field will appear in this font size.
6. **position**. This value identifies the position of the field in a Form object relative to the coordinate system of the window in which it is displayed.

3. Windows

Forms are displayed in either the *Form View window* of the Interface Editor module or the *Input Form window* or *Display Form window* of the Form Use application program. Each window is a descendent of the Prograph system class *Window*, and inherits the attributes and methods of its parent classes.

4. User Interaction

User interaction with Forms is limited to keyboard and mouse input/selection. For data entry, an insertion cursor identifies the active field. The cursor is moved from one field to another either by selecting the destination field with a mouse, or by repeatedly pressing the tab key. The order in which the insertion cursor moves from field to field is determined by the *tab order* of a Form. The tab order is defined in the Interface Editor module at the time a Form is created.

5. Form Methods

Methods for defining and editing Form objects are contained in the Interface Editor module. Methods for data manipulation (entering and displaying data in a Form) are contained in the Form Use application.

6. Data Validation

Data validation (type checking) is performed by methods defined in the Form Use application program. These methods will ultimately be incorporated into the Query Editor by using Prograph's *selective load* feature, which allows loading individual classes from one application into another.

Type checking is possible since each Form is associated with a specific database and database relation. Each field of a Form is further associated with a specific relation attribute (and corresponding attribute data type). This knowledge is obtained from a database definition file which resides on disk and is available to the Interface Editor during Form definition. Type checking takes place as data is entered into a Form by a method that compares the type of the inputted data against the *data type* of the field's associated relation. Data validation methods have not yet been fully implemented.

7. Input Form

Forms used for entering data are displayed in a separate Input Form Window (a descendent of the class *Form Window*). The window is activated in conjunction with a DFQL query. Control objects (icon buttons) are included in the window.

8. Display Form

Forms used for displaying the results of database queries are displayed in a separate *Display Form* window (a descendent of the class *Form Window*). The window is activated in conjunction with a DFQL query. Control objects (icon buttons) are included in the window.

9. Form Creation and Editing

Forms are created and edited in the Interface Editor module's Design Form and Edit Form windows, respectively.

10. Form Use

Forms are used by the Amadeus Query Editor module as part of DFQL queries. Although Forms can be used for both input and data display, there is only one Form class. Depending on the intended use, the Form is opened in either an Input Form or a Display Form window.

a. Input Forms

Figure 13 shows a DFQL operation which opens an Input Form (specified by *Form name*) and inserts data entered via the Form into a relation specified by *relation name*.

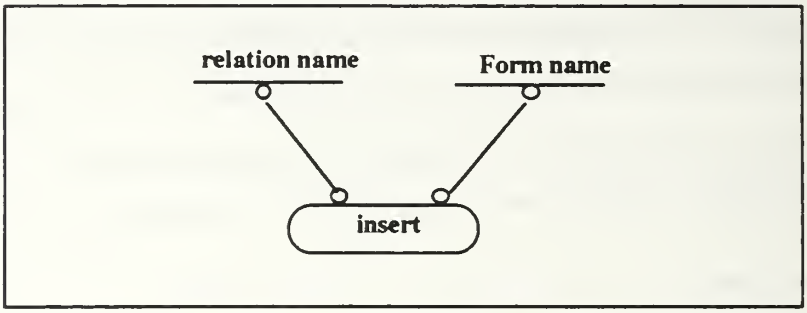


Figure 13: DFQL Use of an Input Form

Forms permit a user to enter data one *tuple* at a time. As data is entered, it is maintained in an attribute called *result* of class *Form Window* as a list of tuples. Each tuple represents the set of data entered into the fields of a Form. This relationship is shown in Figure 14.

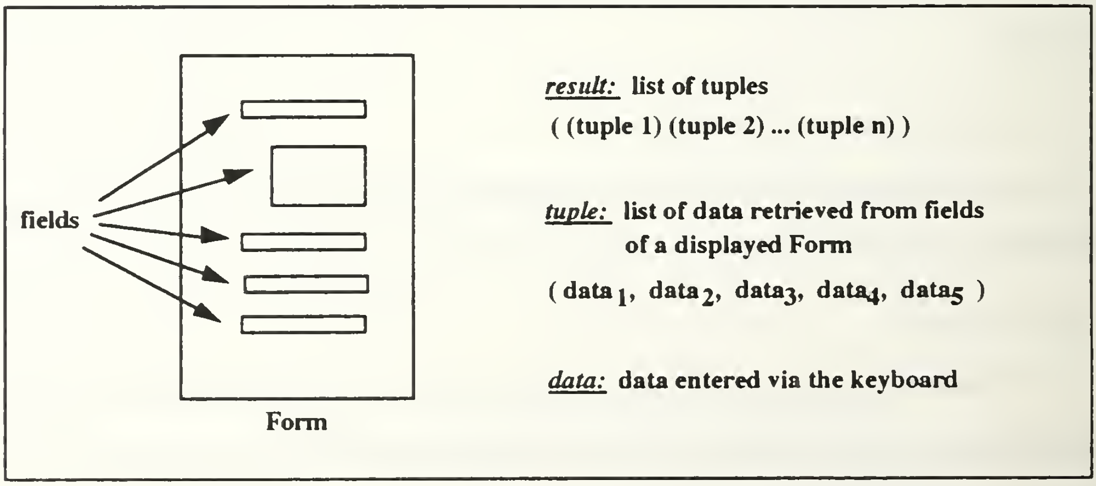

fields
Form
result: list of tuples
((tuple 1) (tuple 2) ... (tuple n))
tuple: list of data retrieved from fields of a displayed Form
(data$_1$, data$_2$, data$_3$, data$_4$, data$_5$)
data: data entered via the keyboard

Figure 14: Representation of Data Entered via an Input Form Window

The ordering of elements (i.e., individual pieces of data) in a tuple corresponds to the order of attributes in a relation's definition. The first element corresponds to the first attribute, the second element to the second attribute, and so on. When a Form object is created, each of its component fields is associated with a specific attribute name. Since the position of each element of a tuple corresponds to a specific attribute position, and each field of a Form corresponds to a specific attribute name, this information can be used to determine the proper position of inputted data in a tuple.

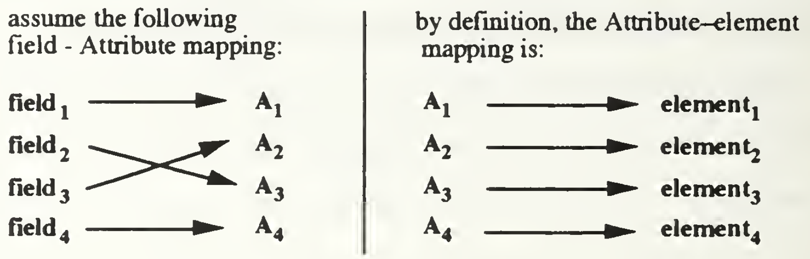
These same relationships are used to support type checking of inputted data. Since each attribute has an associated data type, this value can be compared with the type of the data entered into a Form field. If a type mis-match is identified, the user must correct the data before it can added to a tuple. Figure 15 shows the field-to-element mapping. The notation: $field_1$, $field_2$,$field_3$, etc. in Figure 15 refers to the tab order of a particular field (e.g., field_1 is the first field in the Form's tab order, field_2 is the second, etc.).

Given the following definitions:

relation (A_1, A_2, A_3, A_4)

Form: field $_1$, field$_2$, field$_3$, field$_4$

tuple: (element$_1$, element$_2$, element$_3$, element$_4$)



assume the following field - Attribute mapping:

by definition, the Attribute–element mapping is:

Then data elements will be retrieved from Form fields and stored in tuples as:

(field$_1$, field$_3$, field$_2$, field$_4$)

Tuples, in turn, will be stored in *result* as:

((field$_1$, field$_3$, field$_2$, field$_4$) (field$_1$, field$_3$, field$_2$, field$_4$) ...)

Figure 15: Example of Field-to-Element Mapping

As an example, assume that the Form specified by *Form name* (see Figure 13) contains the following fields: Name, Address, Age, Birthdate and Phone. When the query shown in Figure 13 is processed, an *Input Form* window is opened, and *Form name* displayed as depicted in Figure 16.

If *relation name* has been defined in the database as:

relation name **(Name, Age, Address, Birthdate, Phone)**

where *Name, Age, Address*, *Birthdate* and *Phone* are the attributes of *relation name*, then data entered into *Form name* will be stored in *result* as:

((Name value$_1$, Age value$_1$, Address value$_1$, Birthdate value$_1$, Phone value$_1$)

(Name value$_2$, Age value$_2$, Address value$_2$, Birthdate value$_2$, Phone value$_2$)

.

(Name value$_n$, Age value$_n$, Address value$_n$, Birthdate value$_n$, Phone value$_n$))

When the user closes the *Input Form* window, the data contained in *result* becomes available to the Query Editor module.

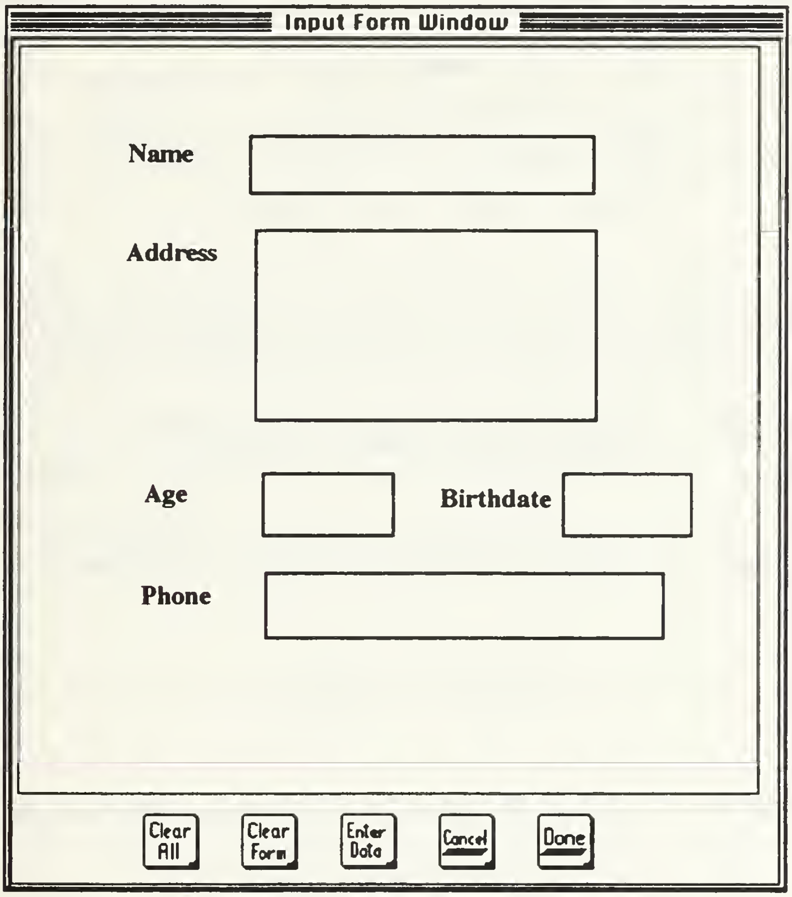


Figure 16: Input Form Window

b. Display Forms

Displaying data from a DFQL query follows a similar logic to that described above for *Input Form*. Figure 17 shows a portion of a DFQL query that might be used to display query results in a *Display Form* window:

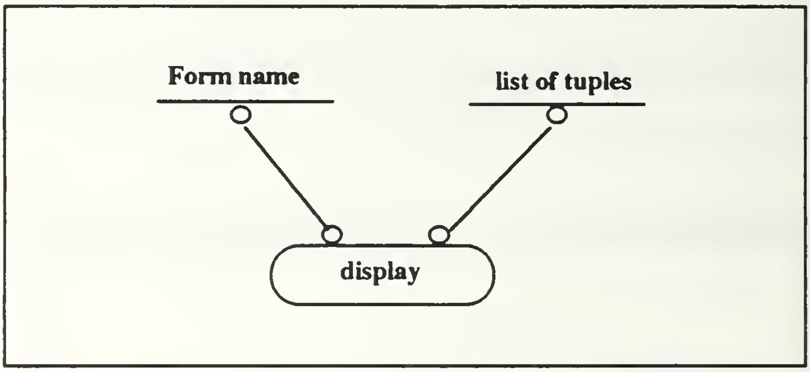


Figure 17: DFQL Use of a Display Form

In this example, *list of tuples* consists of a list of list of strings, and can be expressed as:

((tuple) (tuple) (tuple) ... (tuple))[3]

Since

tuple —> (element$_1$, element$_2$, element$_3$, ... element$_n$)

and

element —> string

then

list of tuples —> ((list of strings) (list of strings) ... (list of strings))

where

(list of strings) —> (string$_1$, string$_2$, string$_3$, ... , string$_n$)

[3] The notation ( ) indicates a list. Thus a list of lists is represented as (( ) ( ) ... ( ))

Each tuple represents a set of data generated as a result of a DFQL query. A DFQL query can produce zero, one or many such tuples. The individual elements (in this case, strings) of a tuple contain the data which is actually displayed in the *Display Form* window.

Just as the field–Attribute mapping is used to build tuples during data entry, the same relationships are used to determine the proper field in which to display each tuple element. Using the definitions and mapping from Figure 15, the mapping from tuple element to Form field can be expressed as:

$$\begin{tikzcd}
\mathbf{element}_1 \arrow[r] & \mathbf{A}_1 \arrow[r] & \mathbf{field}_1 \\
\mathbf{element}_2 \arrow[r] & \mathbf{A}_2 \arrow[dr] & \mathbf{field}_2 \\
\mathbf{element}_3 \arrow[r] & \mathbf{A}_3 \arrow[ur] & \mathbf{field}_3 \\
\mathbf{element}_4 \arrow[r] & \mathbf{A}_4 \arrow[r] & \mathbf{field}_4
\end{tikzcd}$$

Tuples are displayed in a *Display Form* window which contains the Form specified by the left-hand input (*Form name*) to the DFQL operation shown in Figure 17. Data is displayed in the *Display Form* window one tuple at a time. Users can page through the data (list of tuples) using control buttons (*First* tuple, *Last* tuple, *Previous* tuple, *Next* tuple) defined in the *Display Window* class (see Figure 18).

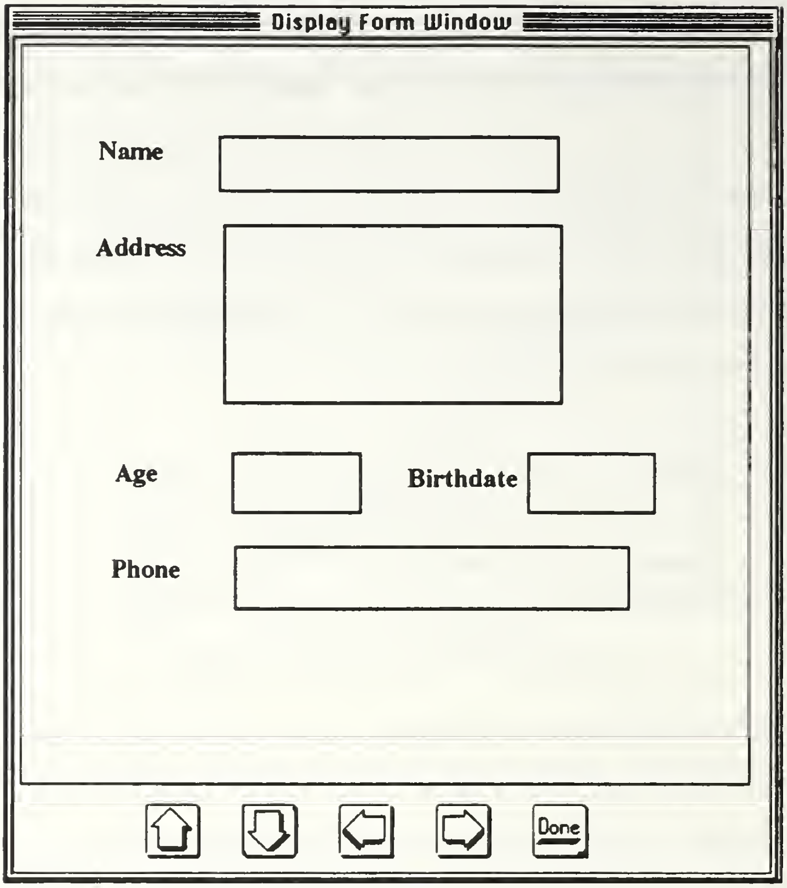


Figure 18: Display Form Window

V. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER RESEARCH

The purpose of this research was to design and implement an Interface Editor module and a Form-based interface for the Amadeus multi-relational database front-end system. The research provided the initial stage of development for a complete Form creation and manipulation capability for Amadeus users.

A. SUMMARY

An extensive literature review was accomplished in which object-oriented programming, the Prograph development environment and Human-Computer Interface design issues were researched. The design and specification for a Form object, a Form creation application (the Interface Editor module) and a Form-based user interface (the Form Use application) were successfully implemented. The feasibility of the Form object and application programs were demonstrated by simulating the Amadeus Query Editor module's use of Forms for data entry and display.

B. CONCLUSIONS

The appropriateness of the user interfaces developed for this thesis can be most effectively measured through formal testing by representative groups of system end-users. Informal testing provided insights into the overall interface design, and lead to improvements in program control functions while identifying areas requiring further development. The Interface Editor, as currently implemented, is limited in scope and functionality. It extends the traditional Macintosh user interface by incorporation of icons for program control. However, the Form objects created by the Interface Editor are limited, at present, to only a single field type. Additionally, the Form Use interface provides a single record-at-a-time view for data entry and the display of DFQL query results. While impacting overall system performance, these limitations are not considered critical. Rather, they provide a point of departure for continued research into enhancing the user interface for the Amadeus system.

C. SUGGESTIONS FOR FURTHER RESEARCH

Further research into the topics presented in this thesis should include, but not be limited to, the following: expansion of the Interface Editor module to allow incorporation of additional window item class objects in a Form field; expansion of the user help facility; implementation of more extensive data validation beyond simple type checking; development of a dynamic Form generation procedure which allows Forms to be generated "on-the-fly" in response to the output of DFQL queries; extending the Form object to allow multiple record-at-a-time viewing; and revising the Interface Editor module to provide a more efficient direct manipulation interface.

1. Expansion of the Interface Editor

The Interface Editor module should permit the inclusion of all window items supported by Prograph as field objects. This capability will provide a more flexible Form object, enhancing the Query Editor module user interface for both data entry and display. Additionally, data representation will become more responsive to user needs, since some information is more appropriately represented as a graphic object, such as radio button sets or check boxes. Expanding the Interface Editor module to meet this capability will require defining new window item subclasses in the same manner as the *IE edit text* subclass of *edit text* was defined. The object-oriented design of the module is well suited for this task.

2. Expansion of the User Help Facility

The Interface Editor module user help facility is very basic in its content and functionality. As currently implemented, only the Interface Editor's main module window has a fully functional help facility. Implementation of the help facility for each module window is still required. Additionally, it may be desirable to develop a context-sensitive help facility which provides more responsive user help for the system.

3. Extension of Data Validation Procedures

The current data validation procedures employed by the Form Use application consist of simple type checking of input data against attribute data types defined for each database. A more robust data validation facility is required which is capable of supporting database unique data types, including formatted data such as *date fields*. Consideration should also be given to the inclusion of range checking (checking for out of range values vice simply verifying the correctness of the associated data type).

4. Dynamic Form Generation.

Currently, all Forms used by the Query Editor must be defined prior to use, requiring a priori knowledge of DFQL queries and their results. This is sufficient for processing data using "canned" queries. However, the ability to dynamically generate a Form based on DFQL query results will provide much greater system flexibility.

5. Extension of the Form Object

The current Form object is only capable of displaying query results one record at a time. This provides a useful, but limited view for the user. The Form object should be extended to permit simultaneous display of multiple records on a single Form.

6. Extension of the Interface Editor Direct Manipulation Interface

The Interface Editor module interface makes use direct manipulation features for dragging and resizing Form fields. However, Form creation and editing takes place in separate modal windows. The use of modal windows restricts, to a certain degree, the user's actions. A more effective interface might be realized if all Form manipulation (including creation and editing) occurs in the Form View window rather than separate modal windows. This would, however, require extensive modifications to the underlying functionality of the interface. User testing may determine whether the efforts required to provided such a feature are worth the cost associated with re-designing the interface.

APPENDIX A

DEVELOPMENT NOTES

The *Interface Editor Module*, *Form object* and *Form Use* application were developed using Prograph version 2.5.2 on a Macintosh IIfx computer with 8 MB of RAM, a 210 MB hard disk and an Apple 21" color monitor. The operating system was System 7.0 (with System 7 Tuner installed). This configuration is considered adequate for developing medium to large-sized Prograph applications.

Version 2.5.2 of Prograph fixes a number of "bugs" present in earlier versions. Additionally, version 2.5.2 replaces two Prograph Extensions and two Window Class methods found in version 2.5.1. For these reasons, the applications and classes developed for this thesis may not work properly on systems running Prograph versions 2.5 or 2.5.1. Version 2.5.2 update patches are available directly from TGS or from commercial electronic bulletin board systems (such as America OnLine and CompuServ).

The *Icon Button* Class (a descendent of *Click Item*) was used to implement the icon buttons used in the applications developed for this thesis. This class is not part of the standard Prograph system release, and is available separately from TGS as part of the *Prograph Goodies Disk*. Icon bar icons were created using ResEdit, and are stored as PICT resources in the resource fork of the Interface Editor application.

A large-screen color monitor (19" or larger) is recommended when developing Prograph applications. It is not unusual to have a dozen or more windows open simultaneously, each displaying a method, case, attribute or class window. The larger screen size allows simultaneous viewing of multiple windows, especially when debugging a program using Prograph's single-step mode for animating data-flow diagrams. Additionally, Prograph uses color coding to distinguish icon and dataflow states, which complicates program development on a black and white or grey-scale monitor. Figure 1 shows a typical 21" screen during program development.

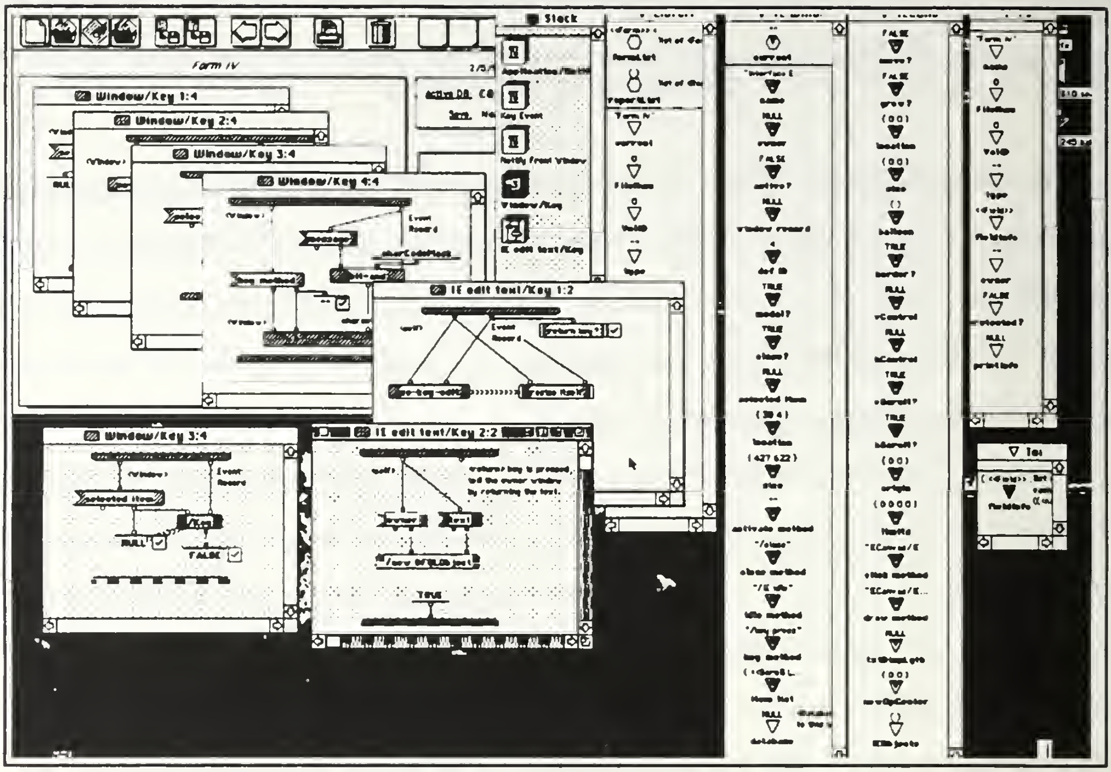


Figure 1. Multiple Windows Open During Application Development

The interface for the Interface Editor module was developed for a 13" (or larger) monitor (ideally 8-bit color, however the effect of color icons will not be degraded when viewed on a grey scale monitor). Although the module will run on a compact Macintosh computer (compact-mac) which is capable of running Prograph[1], the compact-mac's 9" black and white monitor will not correctly display icon buttons (a consequence of attempting to display 8-bit graphics on a 1-bit monitor). Additionally, the Interface Editor main window exceeds the dimensions of a 9" monitor.

Prograph is, essentially, a list processing language. Prior experience with list processing is not necessary in order to use Prograph. However, in order to take full

[1] Minimum system requirements for Prograph are: Macintosh Plus generation with 128K ROM and 1MB of RAM. Prograph can be run from floppy disks, however a hard disk is recommended.

advantage of the language, familiarity with lists and list processing techniques is strongly recommended.

Prograph implements most of the Macintosh Toolbox calls described in Apple Computer Inc's *Inside Macintosh* [Apple85] and *Macintosh Technical Notes* (published separately from *Inside Macintosh*). However, Prograph's accompanying Toolbox documentation (Tutorial and Reference Manuals [TGS90b and TGS90c] and on-line help facilities) are incomplete. Serious Prograph development efforts require access to the complete set of *Inside Macintosh* and *Technical Notes*, the definitive reference sources for Macintosh program development.

APPENDIX B

PROGRAPH BASICS

A. LANGUAGE BASICS

1. Pictorial Representation of the Language

Prograph programs are composed entirely of icons and amplifying text. Figure 1 shows common icons used in constructing Prograph programs.

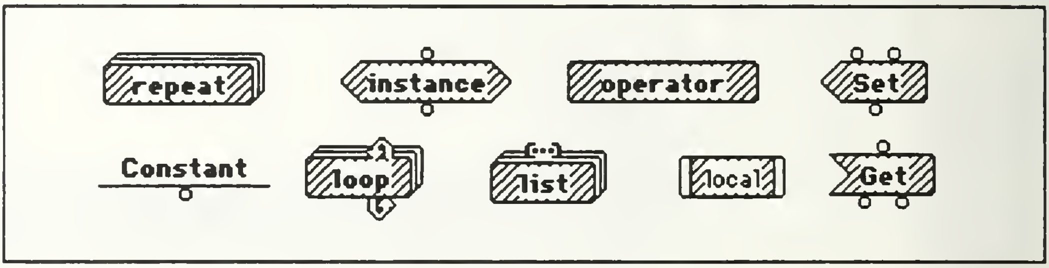


Figure 1. Examples of Prograph Icons

2. Control Structures

Prograph *Control Structures* control the flow of execution within a program. Control structures are composed of icons (either an 'X' or a '√') that are attached to the right-hand side an operator, and are activated on either the success or failure of the associated operation. The default control structure is *success*. Operations *fail* in one of three ways: (1) in a match operation, the items being compared do not match, (2) a Boolean operation returns a FALSE value, or (3) a *FAIL* condition is propagated to a particular operation. Operations may also generate errors under certain conditions, including: type mis-matches, syntax errors, or a specific program condition which can not be satisfied by the particular control structure. Figure 2 shows typical Prograph control structures. An 'X' within a control structure indicates that it is activated if the associated operation fails. A check mark (√) indicates that the control structure is activated if the associated operation within a control structure indicates that it is activated if the associated operation fails. A check mark (√) indicates that the control structure is activated if the associated operation succeeds. Other graphics inside the control structure icon indicate additional action to be taken.

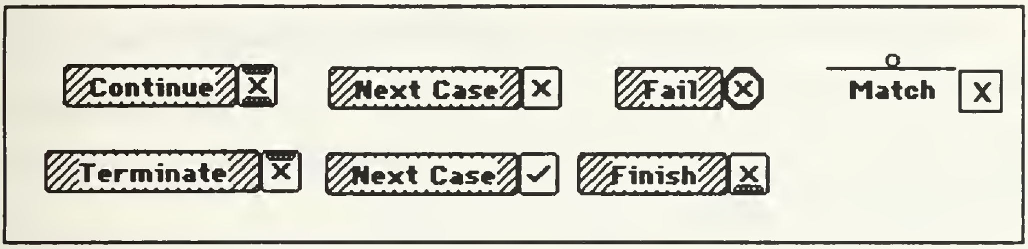
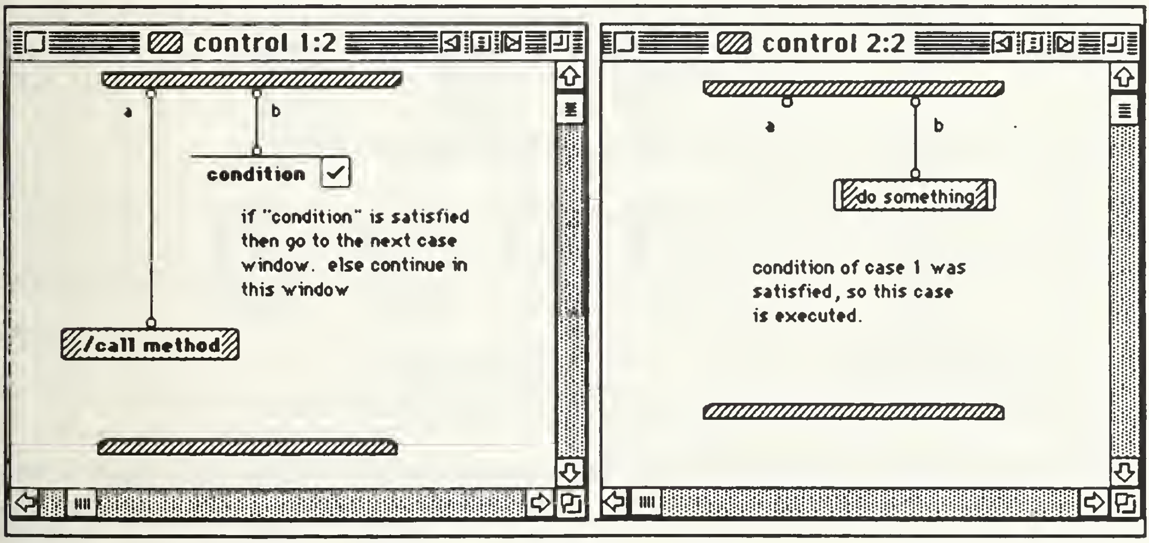


Figure 2: Prograph Control Structure Icons

The most basic Prograph conditional execution format is the *Next Case* with an accompanying match operation or conditional test. Figure 3 depicts a conditional test with a *match on success* control structure which tests for a specific condition to determine which of two case windows will be executed.



if "condition" is satisfied then go to the next case window. else continue in this window

condition of case 1 was satisfied, so this case is executed.

Figure 3: Example of the Next Case on Success Control Structure

3. Classes and Inheritance

Classes of objects, and all inheritance relationships, appear on the screen as trees of icons. The Prograph class system provides a means for constructing a new class from an existing class through inheritance. A Prograph class can inherit from at most one parent. This is referred to as *single inheritance.*

The class icon is a hexagon which is divided into two parts: *attributes* on the left, and *methods* on the right. Double-clicking on the left half of a class icon displays the attributes of the class, while double-clicking on the right half displays the class methods. Figure 4 depicts this relationship.

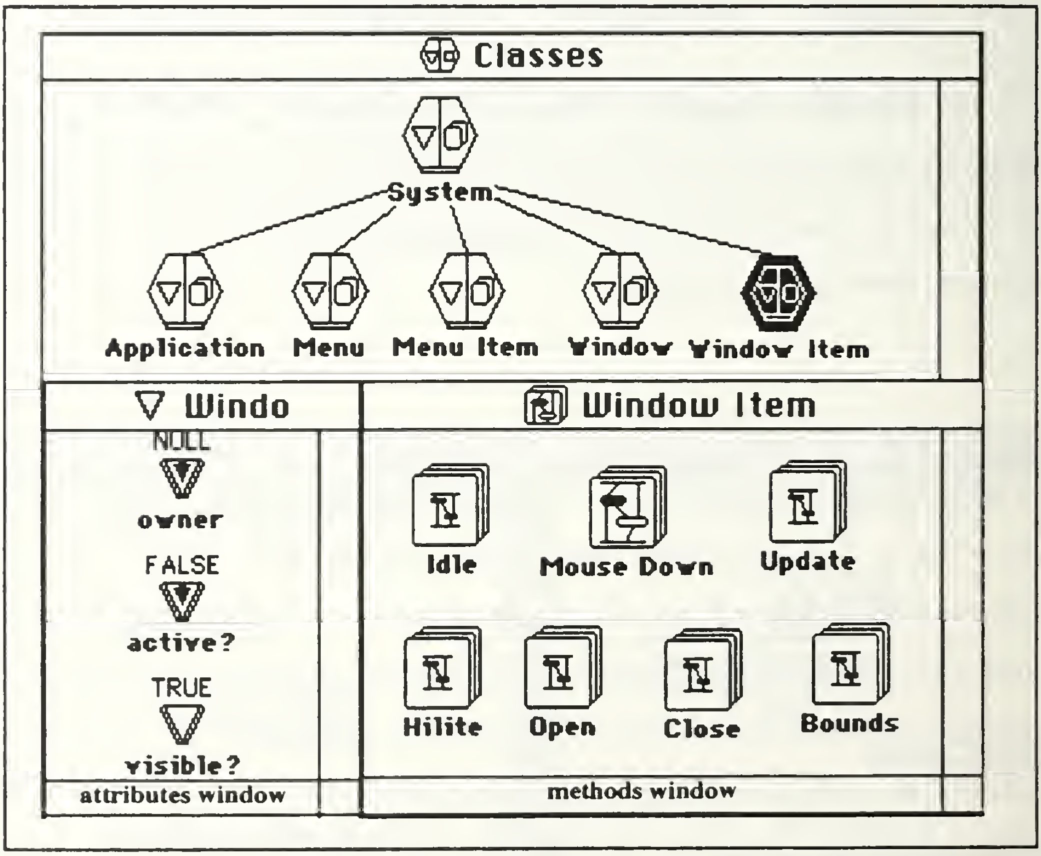


Figure 4: Prograph System Class Icons and Component Parts of "Window Item Class"

4. Attributes

Prograph attributes are displayed in an *Attributes Window*. There are two types of Prograph attributes: *instance* and *class*. An instance attribute may have a different value for each instance of a class. Class attributes, however, have one value for the class as a whole. Therefore, the value of a class attribute is shared by all instances of the class. The attribute icon is a downward pointing triangle.

5. Methods and Cases

A Prograph method consists of a sequence of one or more dataflows, called *cases*. A case consists of an input bar, an output bar, operations and datalinks, Data flows into a case via the input bar, and out through the output bar.

Methods are referenced in one of four ways: *universal, data-determined, explicit and context-determined* (see figure 5). These terms correspond to the terms global, regular, early-bound and self, which are more commonly used in object-oriented programming literature [Wu91c p. 71]. Essentially, the calling format determines where Prograph looks for the referenced method in the class hierarchy.

(1) Universal. This is a call to a global method.

(2) Data-determined. Prograph looks for the referenced method in the class of the object which flows into the leftmost terminal of the method.

(3) Explicit. Prograph looks for the referenced method in the class which is explicitly listed to the left of the "/" in the method icon. If the method is not found in the explicitly listed class, then Prograph uses inheritance links to check ancestor classes for the method.

(4) Context-determined. Prograph looks for the referenced method in the same class as the current method that contains the method referencing operation. This allows a method to send a message to itself.

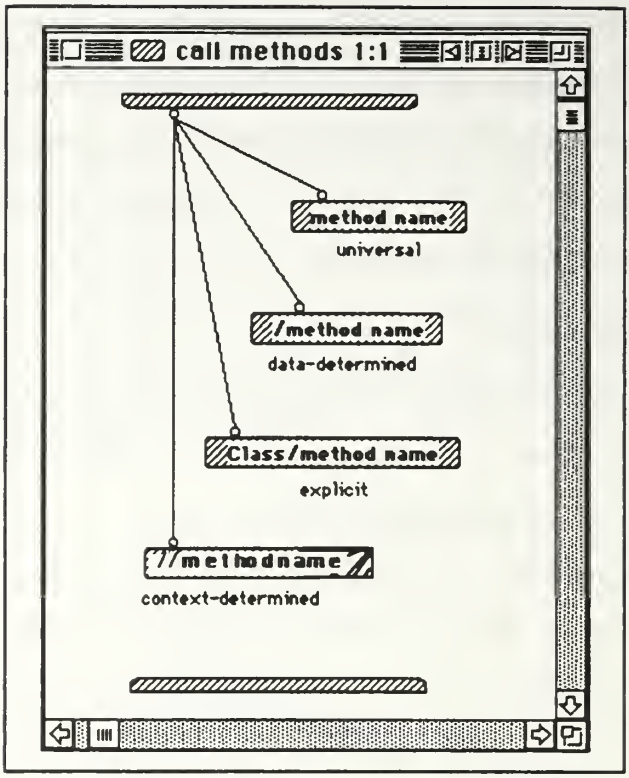


Figure 5: Method Calling Formats

6. Operations

An operation is the basic executable component of a case. Operations have a name, zero or more inputs, zero or more outputs and a distinctive icon. Data flows into an operation through terminals located on the top of the operation icon, and out through roots located on the bottom of the icon. Prograph provides a special icon, called a *synchro link* which forces a specific execution *order* on a pair of operations (see figure 6). However, the synchro link does not guarantee that the operations will execute consecutively, only that one will execute before the other. [TGS90c p. 7] In the example shown below, A will execute before B. However, there is no guarantee that B will execute *immediately* after A, since there is no way to determine when C will execute.

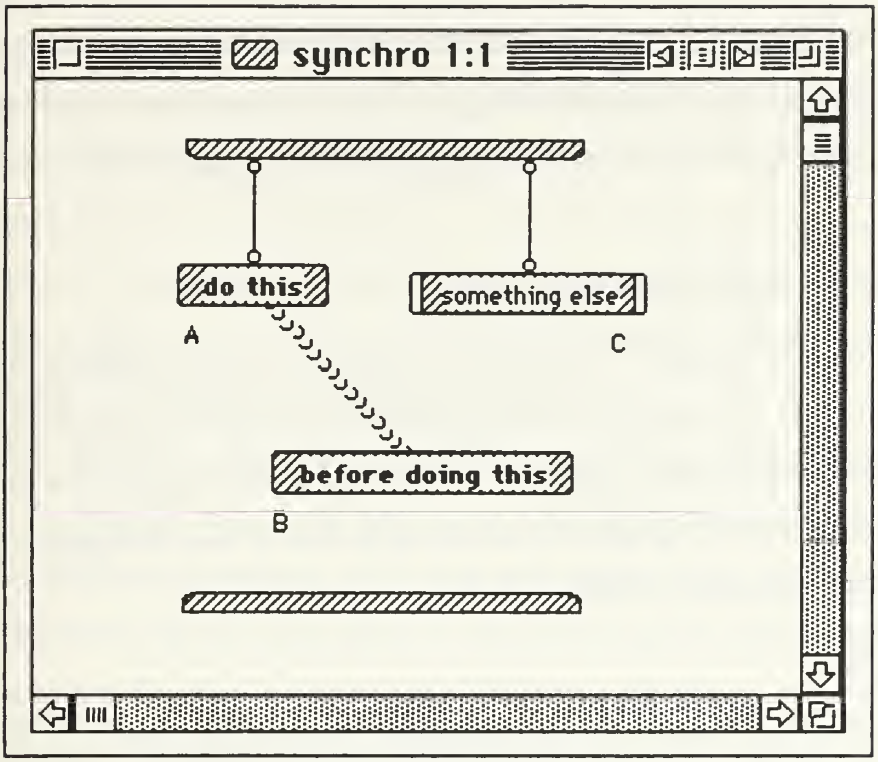


Figure 6: Synchro Link

7. Message Passing

Message passing in Prograph is similar to most other object-oriented languages. Some differences occur, however, because of the dataflow nature of the Prograph language. Essentially, in Prograph objects flow into operations to initiate actions. In a “standard” object-oriented programming language, a stationary object sends a message to another stationary object. Although the models are somewhat different, the basic concepts are the same. [TGS90b p. 93]

8. Primitives

Prograph primitives are calls to compiled methods, and are categorized into sixteen groups, including: Application, Bit, Data, File, Graphics, Instances, Interpreter Control, I/O, Lists, Logical/Relational, Math, Memory, Strings, System, Text and Type.

Primitives comprise the kernel of Prograph's functionality. Unlike other object-oriented programming languages, Prograph primitives do not belong to any class. This, and the fact that the language supports regular data types such as ***string, integer, Boolean*** and ***real*** make Prograph a hybrid object-oriented programming language. [Wu91c p. 72]

B. THE PROGRAPH ENVIRONMENT

The Prograph language is seamlessly integrated with the Prograph development environment. An editor provides a visual interface for creating and modifying programs, while an interpreter contains features which allow dataflow diagrams to be displayed during execution, in effect graphically animating the flow of data throughout a program as each operation is executed [TGS90a p. 21].

1. Editor

The Prograph editor is context sensitive, so syntax errors are caught at the time they are created, eliminating the need for a traditional debugger. During program execution, run-time errors are flagged, program execution is halted and the appropriate dataflow diagram displayed. This enables the user to correct the error and immediately resume execution. An on-line help system is also available and is fully integrated into the editor.

2. Interpreter

The Prograph interpreter is highly interactive. Program execution may be paused at any point and dataflow diagrams and data values examined, allowing simultaneous execution and editing of applications. Additionally, program execution may be traced step by step, allowing the flow of data through a program to be traced visually. If a dataflow diagram is changed while execution is paused, the interpreter backs up to the change and continues execution from that point.

C. COMPILER

The Prograph compiler generates stand-alone application programs, and allows linking to modules developed with other programming languages such as MPW C and Think C. The compiler also includes an intelligent *Project Manager* which keeps track of the files needed to build a particular application. The Project Manager selects only the code actually required when building a stand-alone application and informs the user of any missing code. If the compiler detects an error in a Prograph file, the user can enter the editor/interpreter to see the operation that generated the error.

A certain amount of overhead is normally introduced when creating stand-alone applications. In Prograph, stand-alone applications which do not use system classes require an additional 50Kbytes of overhead, while those with system classes require an additional 130Kbytes. However, the execution speed of compiled Prograph code is, on the average, 15 times faster than the same interpreted code [TGS90a p. 33-36].

APPENDIX C

DATABASE BASICS

A. DEFINITIONS

1. Database

In a general sense, a *database* is a collection of related, recordable facts that have implicit meaning. To be more precise, however, a database may be defined as a "shared collection of inter-related data designed to meet the varied information needs of an organization" [Falby91 p. 14]. Databases have the following properties [Elmasri89 p. 3-4]:

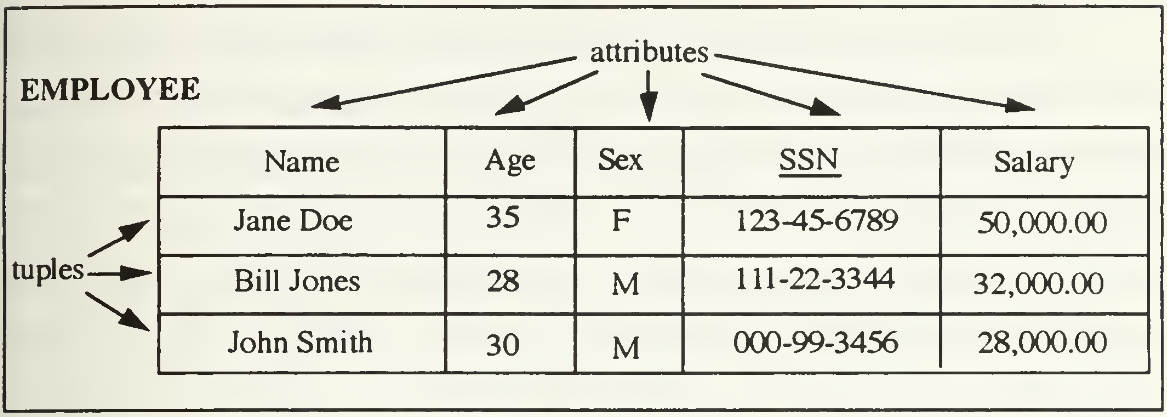
1. Logically coherent collection of data with some inherent meaning.
2. Designed, built and populated with data for a specific purpose.
3. Represents some aspect of the real world (referred to as the *mini-world*).

2. Database Management System (DBMS)

A DBMS is a general-purpose software system for defining, constructing and manipulating a database.

3. Relational model

The relational model for Database Management Systems was first introduced in 1970. The model is founded solidly on mathematical principles and provides simple, uniform data structures. The relational model represents a database as a collection of tables. Each row in a table represents a collection of related data values which can be interpreted as a fact describing an entity or a relationship instance. The table name, and the names of the table columns, provide additional meaning to the values in each row of the table. Each row in a relational database relation is called a *tuple*, and each column title is called an *attribute*. The Table itself is referred to as a *relation*. [Elmasri89 p. 135-137]. Figure 1 shows a relation, named EMPLOYEE, from a relational database.



Name	Age	Sex	SSN	Salary
Jane Doe	35	F	123-45-6789	50,000.00
Bill Jones	28	M	111-22-3344	32,000.00
John Smith	30	M	000-99-3456	28,000.00

Figure 1: Table from a Relational Database

4. Data Manipulation Languages & Database Query Languages

Database Management Systems can provide two types of *Data Manipulation Languages* (DML) which allow users to manipulate data in the database: *high-level* or *non-procedural* and *low-level* or *procedural*. High-level DML's can be used either as stand-alone languages or can be embedded in a general-purpose programming language. When used as a stand-along language, high-level DML statements are entered interactively from a terminal by a DBMS user, and the DML is called a *database query language*. Low-level DML's must always be embedded in a general-purpose programming language. Casual DBMS users normally use a high-level query language to manipulate the database, while programmers use a DML which has been embedded in a general-purpose programming language.

APPENDIX D

INTERFACE DESIGN GOALS AND CONCERNS

The goal of interface design should be to empower people, leading to an increase in user experience, productivity and creativity [Dertouzos90, Butler90 p. 349]. Users are empowered when they have "a clear predictive model of system performance and a sense of mastery, control, and accomplishment" [Don92 p. 69]. To truly empower a user, however, the interface designer must fully understand the difference between "efficiency" in terms of computer science and "productivity" in terms of getting more value from work [Winograd90]. A well designed interface should be practically transparent, allowing users to concentrate directly on the task at hand [Shneiderman92].

A. THE DESIGN PROCESS

The design process should typically begin with an understanding of the system's intended users to develop a *user profile*. It is not uncommon to characterize system users into different groups or classes. This may result in different design goals for each user class. For example, classifying users as *novice, intermediate* and *advanced* will require unique features for each user class. [Schneiderman92 pages 145-148]

After completing the user profile, task analysis should be conducted to identify the functionality required of the proposed system. Good task analysis means continual user testing, starting as soon as the work begins. Key elements of the design should emerge from the task analysis, rather than being shaped to fit the results of the user testing [Norman in Laurel90 page 9].

Change is important to good interface design. Procedures that allow incorporation of changes, up to a point, are considered critical to successful design. Ideally, this should include an Iterative design methodology involving user testing and prototyping tools to rapidly produce non-functional and functional models of the system to elicit user feedback at key stages of the design process [Mountford90]. Iterative design is a process of identifying a problem area, modifying the appropriate portion(s) of the application, and re-testing. The iterative process continues until a decision is made to accept the prototype. Most problems will disappear by the second or third iteration [Tognizzini92 page 86].

The cost of user testing has been a topic of debate in recent years. Historically, elaborate test scenarios, complete with high-tech equipment for recording and analyzing user actions and perceptions, reinforced the belief that user testing was a costly undertaking. Newer approaches described in [Tognazzini91 pp. 79-91] stress simplicity and common sense, promising to make user testing more acceptable both in terms of cost and end-user productivity. These approaches include:

1. Develop test scenarios that incorporate situations that users may face, then build prototypes that enable evaluation of the situations. Two types of prototypes can be built: *horizontal* and *vertical*. Horizontal prototypes allow testing overall design concepts by displaying all or most of an application's menus, windows and dialogs without going into depth in any one area. Vertical prototypes allow greater examination of specific parts of an application, and are used when new design concepts and/or technology are involved.
2. Prototypes can and should be created in a matter of days, not weeks or months. A wide variety of prototyping tools are available which greatly aid prototype development.
3. Choose test subjects carefully. In the initial stages, user testing should focus on interface issues. Content testing (testing the actual performance and applicability of a system - such as an accounting package or CAD package) is generally not a concern during iterative design testing. Studies have shown that a maximum of three test subjects per design iteration is sufficient [Tognazzini page 82]. Any more than that will simple serve to confirm problems already identified. Each iteration should involve new test subjects [Tognazzini page 270]. Once the interface design has been finalized, validation testing should be conducted. Validation testing requires a

carefully chosen representative sample of the intended user population, and may be applied to late alpha or early beta versions of the application.

4. Apply a standardized methodology for observing test subjects. Apple Computer uses a methodology called *User Observation Through Thinking Out Loud* [Gomoll in Laurel90 pages 85-90]. This process does not yield statistical results. Rather, it allows an observer to identify areas where test subjects are having problems using the product, and provides information necessary to improve it.

B. FACTORS THAT INFLUENCE INTERFACE DESIGN

In general, interface design problems can be easy to describe, but extremely difficult to actually solve [Erickson91]. A good deal of thought, insight, ingenuity, understanding of the original problem and knowledge of the end-user are required; however this still may not be enough to satisfy everybody that a solution is good.

1. Lack of Standardized Methodology

There is no standard method of looking at interface design. This lack of standardization can lead to confusion in design, testing and implementation of user interfaces. Russel summarizes this problem in [Russel92 p. 71-72]:

> As the state of graphical user interface design theory continues to mature from its beginnings in the mid-eighties, emphasis on reducing the burden of interface programming has become more prevalent. Across the many popular computer platforms used professionally today, a myriad of interface building packages have been designed with this concern in mind. Currently, however, no standard design methodology exists and more importantly, no construction package has emerged that significantly reduces programming effort while still providing the application interface programmer with full control over the design process... .

2. Compromise

Interface design is largely a compromise [Erickson91]. Software and hardware must complement each other; an elegant software solution implemented on the wrong platform will not necessarily function the way the designers intended. Human factors must also be addressed (i.e., the interface must take into account human capabilities and limitations). Sophisticated features of an interface may not be practical if the human user cannot take advantage of them.

Additionally, an interface design team is usually inter-disciplinary. Different disciplines have different priorities, thinking styles and values. When people from different disciplines get together, values tend to collide. [Kim in Laurel90 page 32] Thus, competing concerns must constantly be balanced in order to arrive at a mutually acceptable solution.

3. Appropriateness of the Interface

A common design error is to include too much functionality into a system, producing a number of undesirable side-effects, including: excessive code to test, debug and maintain, more complicated user manuals and help facilities and a steeper learning curve and potentially higher cognitive loading on the user. Problems also arise if insufficient functionality is included. In such cases, users may become frustrated because of a perception that the system does not adequately support specific functions or features. [Schneiderman92]

4. Pressure to Produce

Interface designers are often under a great deal of pressure to ship a product quickly, resulting in a tendency to accept the first promising design idea and forego proven testing and development procedures [Mountford90].

APPENDIX E

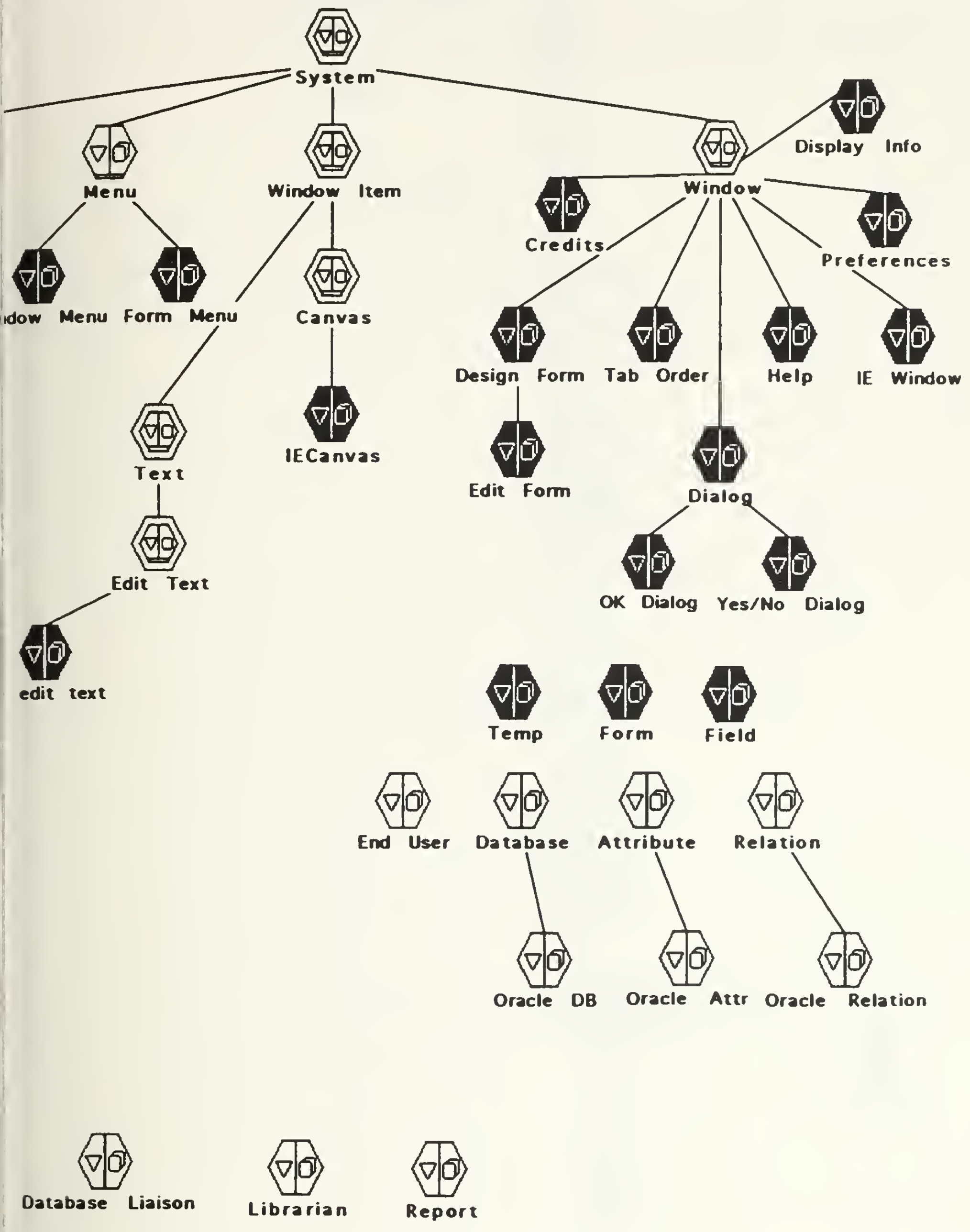
PROGRAM LISTINGS FOR:

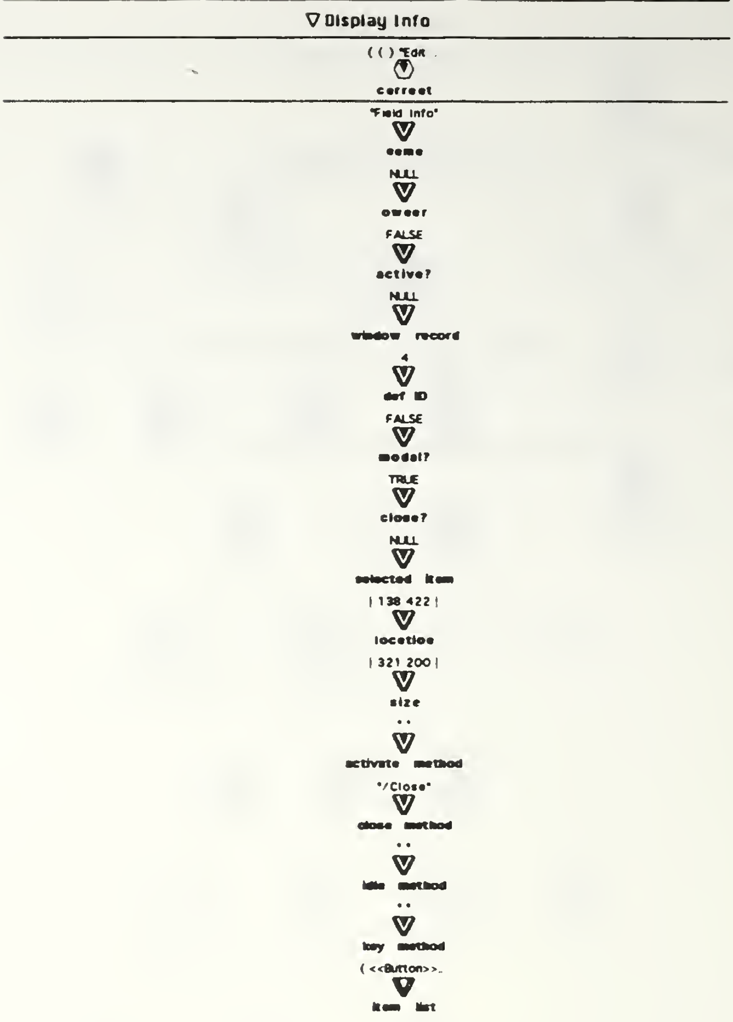
THE INTERFACE EDITOR MODULE

AND

THE FORM USE APPLICATION

Classes


System
Menu
Window Item
Window
Display Info
Credits
Preferences
dow Menu
Form Menu
Canvas
Design Form
Tab Order
Help
IE Window
Text
IECanvas
Edit Form
Dialog
Edit Text
OK Dialog
Yes/No Dialog
edit text
Temp
Form
Field
End User
Database
Attribute
Relation
Oracle DB
Oracle Attr
Oracle Relation
Database Liaison
Librarian
Report



∇ Display Info

(( ) "Edit .

correct

"Field Info"

name

NULL

owner

FALSE

active?

NULL

window record

4

def ID

FALSE

modal?

TRUE

close?

NULL

selected item

| 138 422 |

location

| 321 200 |

size

• •

activate method

"/Close"

close method

• •

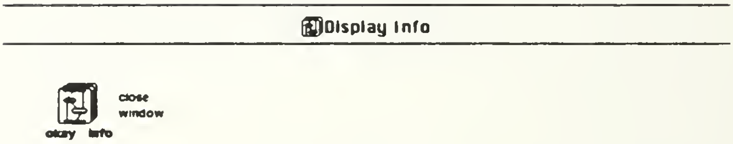
idle method

• •

key method

(<<Button>>..

item list

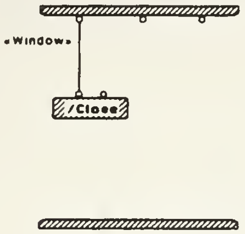


Display Info

okay info

close window

▨Display info/okay info 1:1



∇Credits

current

"Credits"
∇
name

NULL
∇
owner

FALSE
∇
active?

NULL
∇
window record

10
∇
def ID

TRUE
∇
modal?

FALSE
∇
close?

NULL
∇
selected item

| 86 147 |
∇
location

| 225 334 |
∇
size

• •
∇
activate method

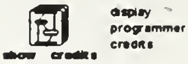
"/Close"
∇
close method

• •
∇
idle method

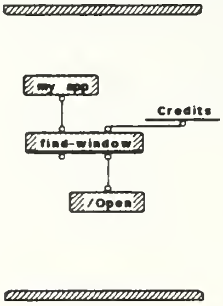
• •
∇
key method

(<<Pict But..
∇
item list

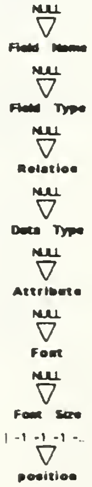
Credits



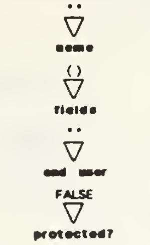
Credits/show credits 1:1



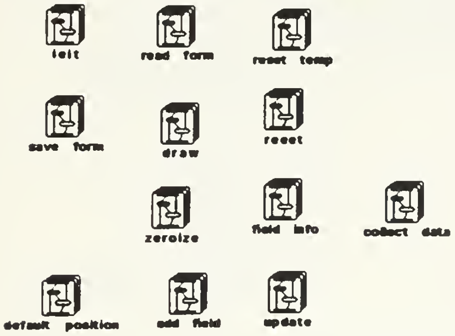
Field



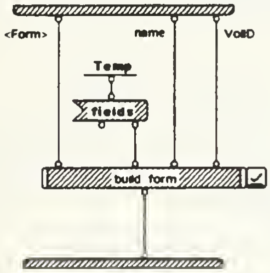
∇Form



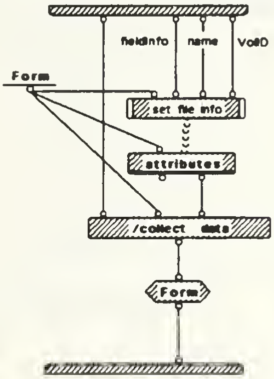
Form



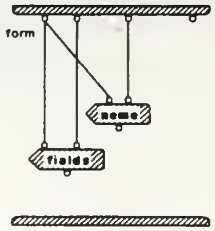
▨Form/save form 1:1



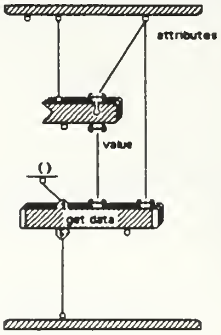
▨Form/save form 1:1build form 1:1



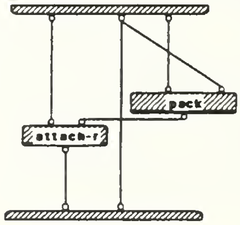
▨Form/save form 1:1build form 1:1set file info 1:1



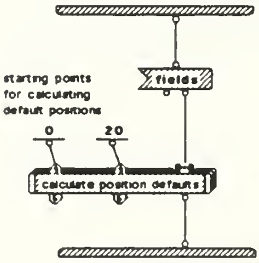
▨Form/collect data 1:1



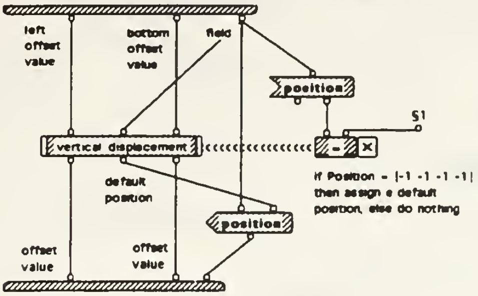
▨Form/collect data 1:1get data 1:1



▨Form/default position 1:1



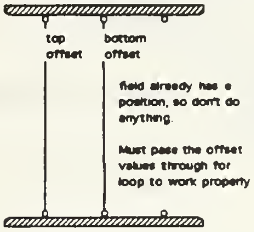
▨Form/default position 1:1calculate position defaults 1:2



If Position = {-1 -1 -1 -1} then assign a default position, else do nothing

$1. {-1 -1 -1 -1}

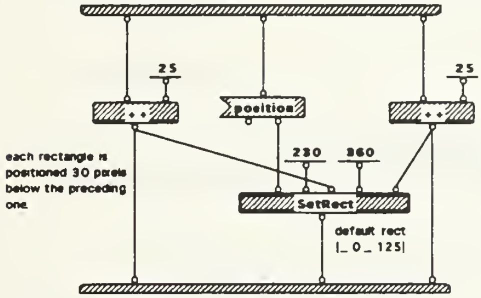
▨Form/default position 1:1calculate position defaults 2:2



field already has a position, so don't do anything.

Must pass the offset values through for loop to work properly

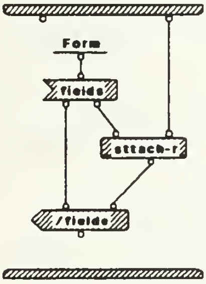
▨Form/default position 1 1calculate position defaults 1 2vertical displacement 1 1



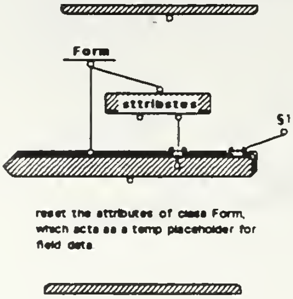
each rectangle is positioned 30 pixels below the preceding one.

default rect {_ 0 _ 125}

▨Form/add field 1:1



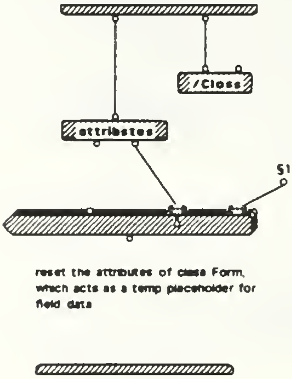
▨Form/reset 1:1



reset the attributes of class Form, which acts as a temp placeholder for field data

§1 ("" () "" FALSE)

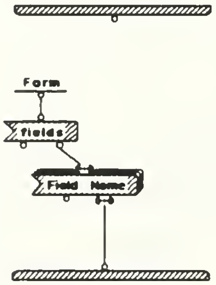
▨Form/reset temp 1:1



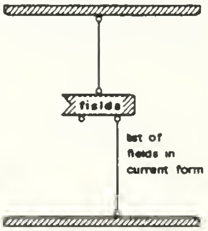
reset the attributes of class Form, which acts as a temp placeholder for field data

§1 ("" 0 0 "" ( ) "" FALSE NULL)

▨Form/read form 1:1

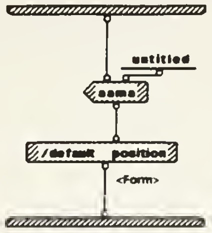


▨Form/field info 1:1

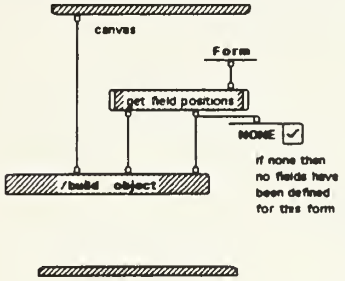


list of fields in current form

▨Form/init 1:1

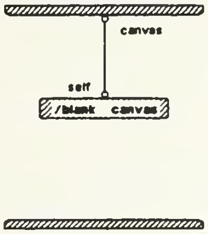


▨Form/draw 1:2

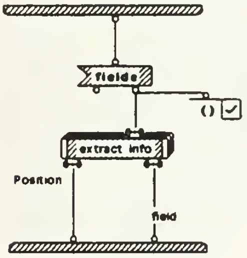


if none then no fields have been defined for this form

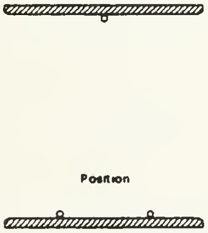
▨Form/draw 2:2



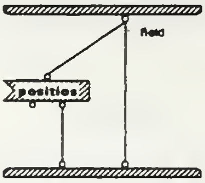
▨Form/draw 1:2get field positions 1:2



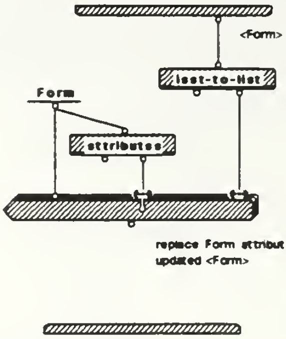
▨Form/draw 1:2get field positions 2:2



▨Form/draw 1:2get field positions 1:2extract info 1:1

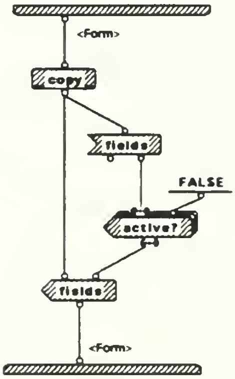


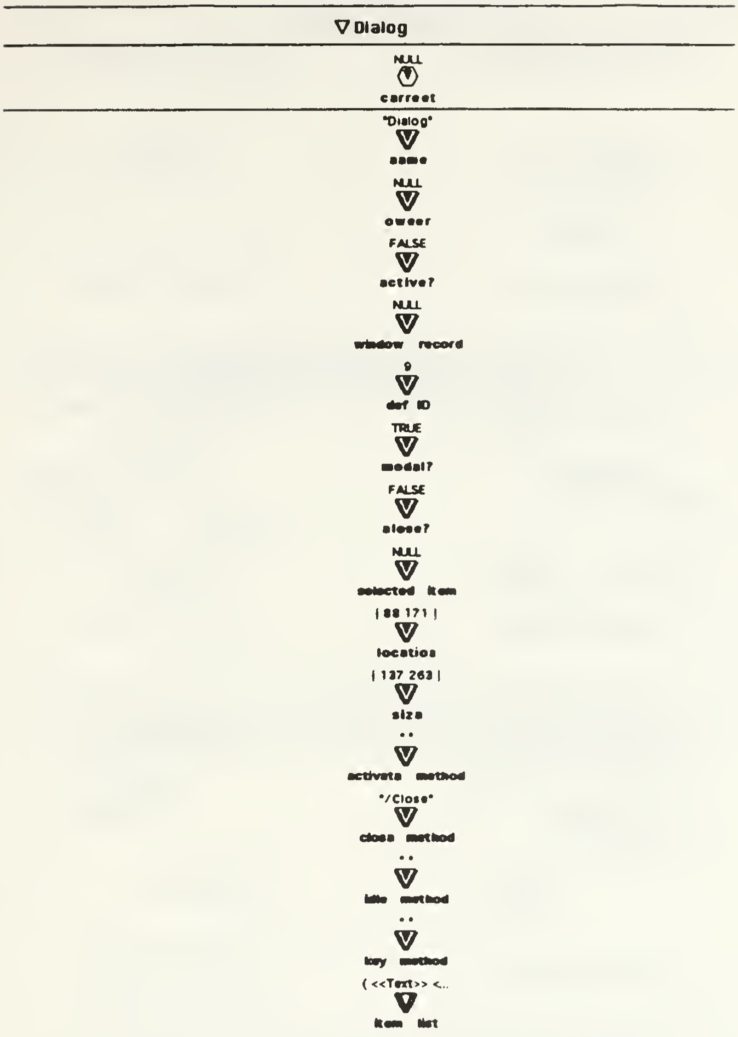
▨Form/update 1:1

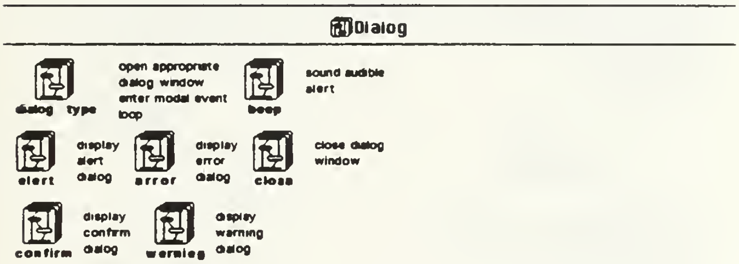


replace Form attributes with
updated <Form>

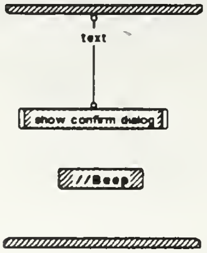
▨Form/zeroize 1:1



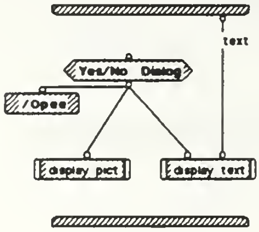

▽ Dialog
NULL
current
"Dialog"
name
NULL
owner
FALSE
active?
NULL
window record
9
def ID
TRUE
modal?
FALSE
close?
NULL
selected item
(88 171)
location
(137 263)
size
• •
activate method
"/Close"
close method
• •
idle method
• •
key method
(<<Text>> <..
item list


Dialog
dialog type
open appropriate dialog window enter modal event loop
beep
sound audible alert
alert
display alert dialog
error
display error dialog
close
close dialog window
confirm
display confirm dialog
warning
display warning dialog

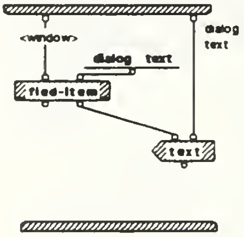
▨Dialog/confirm 1:1



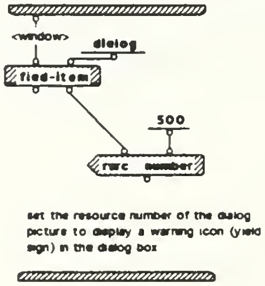
▨Dialog/confirm 1:1show confirm dialog 1:1



▨Dialog/confirm 1:1show confirm dialog 1:1display text 1:1

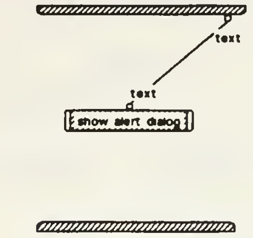


▨Dialog/confirm 1:1show confirm dialog 1:1display pict 1:1

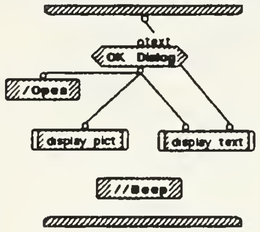


set the resource number of the dialog picture to display a warning icon (yield sign) in the dialog box

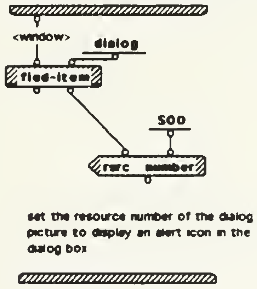
▨Dialog/alert 1:1



▨Dialog/alert 1:1show alert dialog 1:1

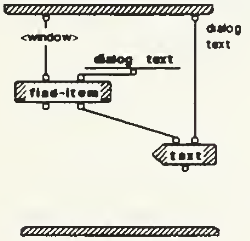


▨Dialog/alert 1:1show alert dialog 1:1display pict 1:1

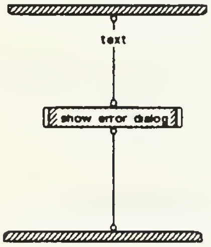


set the resource number of the dialog picture to display an alert icon in the dialog box

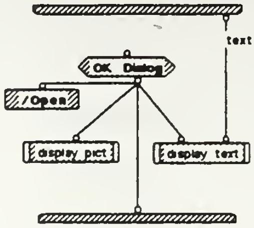
▨Dialog/alert 1:1show alert dialog 1:1display text 1:1



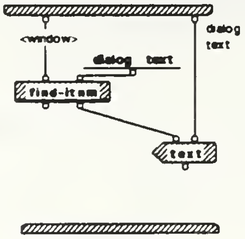
▨Dialog/error 1:1



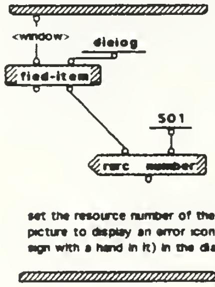
▨Dialog/error 1:1show error dialog 1:1



▨Dialog/error 1:1show error dialog 1:1display text 1:1

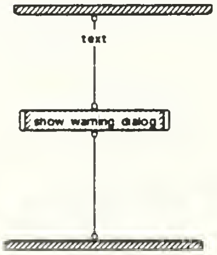


▨Dialog/error 1:1show error dialog 1:1display pict 1:1

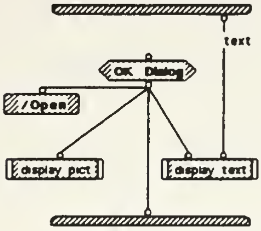


set the resource number of the dialog picture to display an error icon (stop sign with a hand in it) in the dialog box

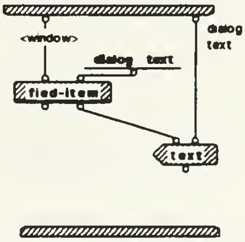
▨Dialog/warning 1:1



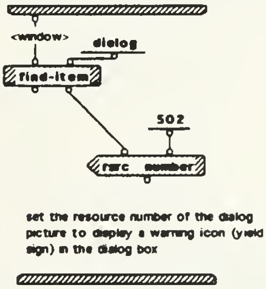
▨Dialog/warning 1:1show warning dialog 1:1



▨Dialog/warning 1:1show warning dialog 1:1display text 1:1

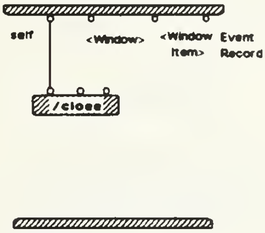


▨Dialog/warning 1:1show warning dialog 1:1display pict 1:1

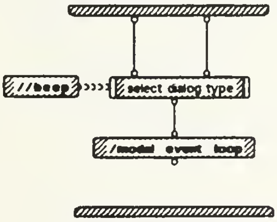


set the resource number of the dialog picture to display a warning icon (yield sign) in the dialog box

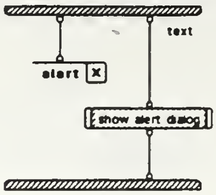
▨Dialog/close 1:1



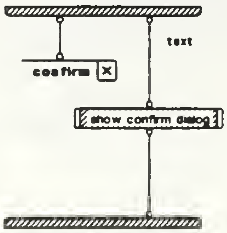
▨Dialog/dialog type 1:1



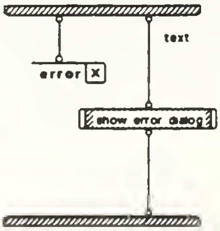
▨Dialog/dialog type 1:1select dialog type 1:4



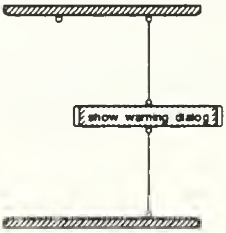
▨Dialog/dialog type 1:1select dialog type 2:4



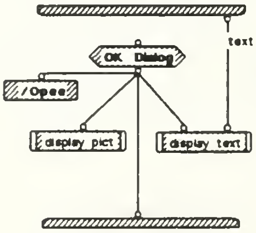
▨Dialog/dialog type 1:1select dialog type 3:4



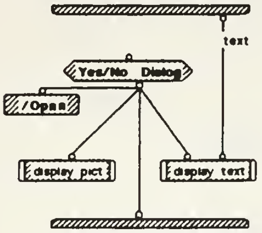
▨Dialog/dialog type 1:1select dialog type 4:4



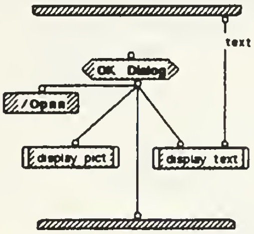
▨Dialog/dialog type 1:1select dialog type 1:4show alert dialog 1:1



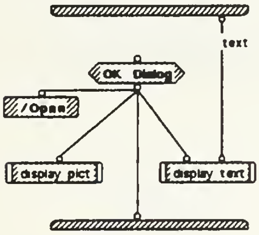
▨Dialog/dialog type 1:1select dialog type 2:4show confirm dialog 1:1



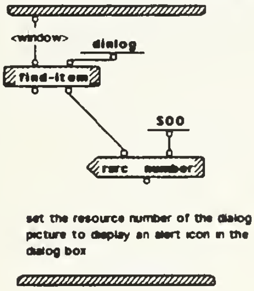
▨Dialog/dialog type 1:1select dialog type 3:4show error dialog 1:1



▨Dialog/dialog type 1:1select dialog type 4:4show warning dialog 1:1

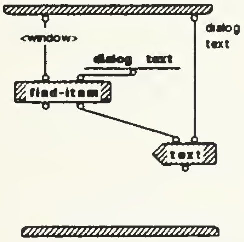


▨Dialog/dialog type 1:1select dialog type 1:4show alert dialog 1:1display pict 1:1

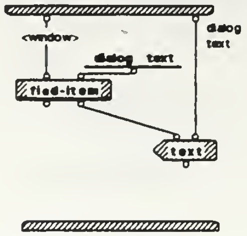


set the resource number of the dialog picture to display an alert icon in the dialog box

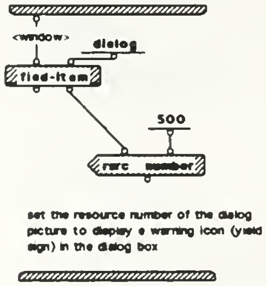
▨Dialog/dialog type 1.1select dialog type 1.4show alert dialog 1:1display text 1 1



▨Dialog/dialog type 1:1select dialog type 2:4show confirm dialog 1:1display text 1:1

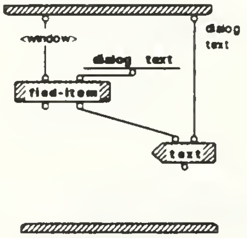


▨Dialog/dialog type 1:1select dialog type 2:4show confirm dialog 1:1display pict 1:1

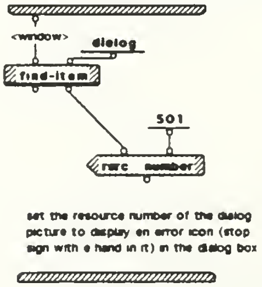


set the resource number of the dialog picture to display a warning icon (yield sign) in the dialog box

▨Dialog/dialog type 1:1select dialog type 3:4show error dialog 1:1display text 1:1

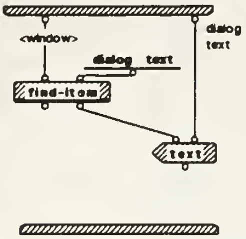


▨Dialog/dialog type 1:1select dialog type 3:4show error dialog 1:1display pict 1:1

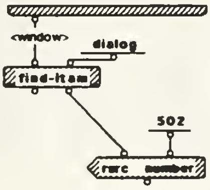


set the resource number of the dialog picture to display an error icon (stop sign with a hand in it) in the dialog box

▨Dialog/dialog type 1:1select dialog type 4 4show warning dialog 1:1display text 1:1

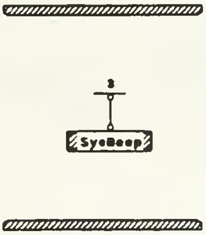


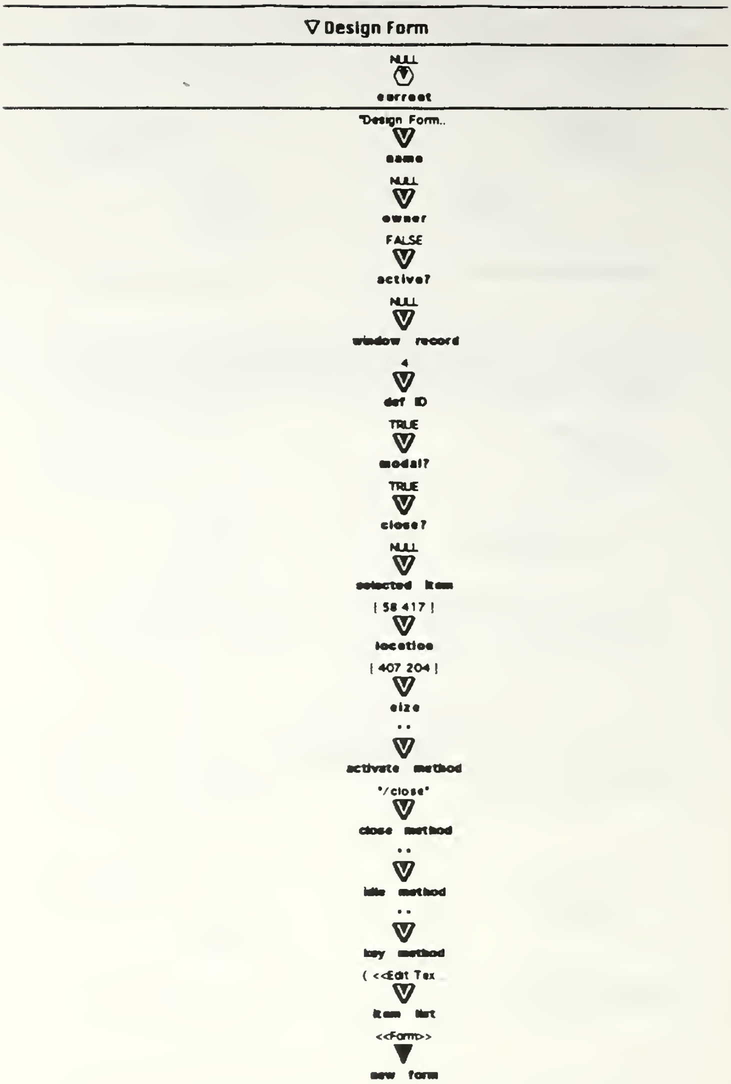
▨Dialog/dialog type 1:1select dialog type 4:4show warning dialog 1:1display pict 1 1



set the resource number of the dialog picture to display a warning icon (yield sign) in the dialog box

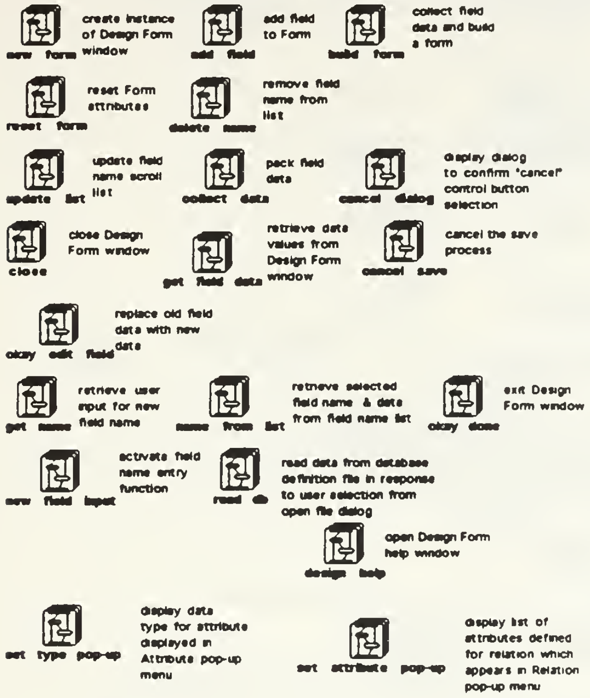
▨Dialog/beep 1:1





```
∇ Design Form

NULL
current

'Design Form..
name
NULL
owner
FALSE
active?
NULL
window record
4
def ID
TRUE
modal?
TRUE
close?
NULL
selected item
( 58 417 )
location
( 407 204 )
size
• •
activate method
"/close"
close method
• •
idle method
• •
key method
( <<Edit Tex
item list
<<Form>>
new form
```

Design Form

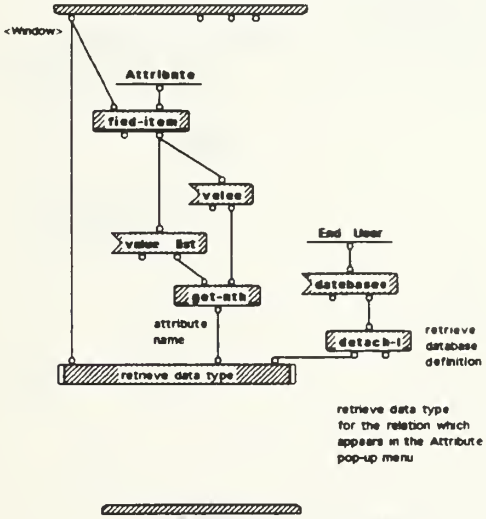


new form	create instance of Design Form window
add field	add field to Form
build form	collect field data and build a form
reset form	reset Form attributes
delete name	remove field name from list
update list	update field name scroll list
collect data	pack field data
cancel dialog	display dialog to confirm "cancel" control button selection
close	close Design Form window
get field data	retrieve data values from Design Form window
cancel save	cancel the save process
okay edit field	replace old field data with new data
get name	retrieve user input for new field name
name from list	retrieve selected field name & data from field name list
okay done	exit Design Form window
new field input	activate field name entry function
read db	read data from database definition file in response to user selection from open file dialog
design help	open Design Form help window
set type pop-up	display data type for attribute displayed in Attribute pop-up menu
set attribute pop-up	display list of attributes defined for relation which appears in Relation pop-up menu

▨Design Form/design help 1:1

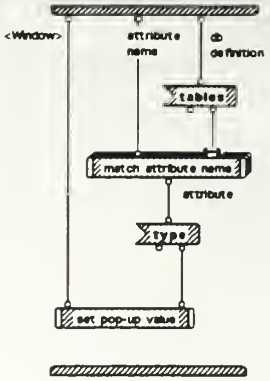
<Window>
<Pict Button>
Event Record

▨Design Form/set type pop-up 1:1

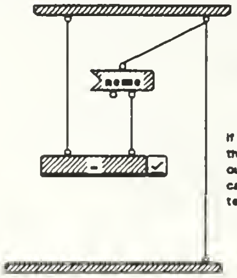


retrieve data type for the relation which appears in the Attribute pop-up menu

▨Design Form/set type pop-up 1:1retrieve data type 1:1

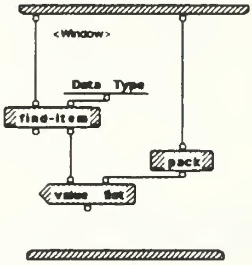


▨Design Form/set type pop-up 1:1retrieve data type 1:1match attribute name 1:1

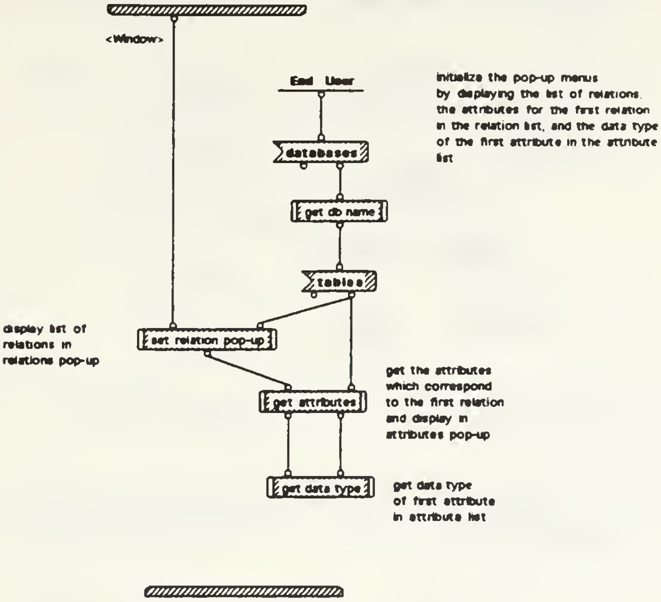


If a match is found, the name will be output on the case output bar terminal

▨Design Form/set type pop-up 1:1retrieve data type 1:1set pop-up value 1:1



▨Design Form/read db 1:1



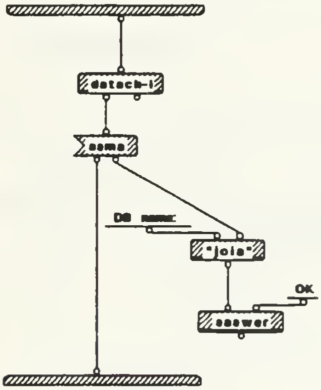
initialize the pop-up menus by displaying the list of relations, the attributes for the first relation in the relation list, and the data type of the first attribute in the attribute list

display list of relations in relations pop-up

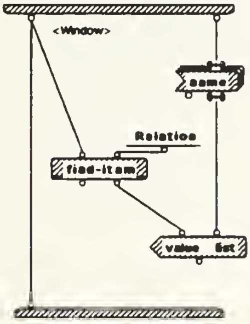
get the attributes which correspond to the first relation and display in attributes pop-up

get data type of first attribute in attribute list

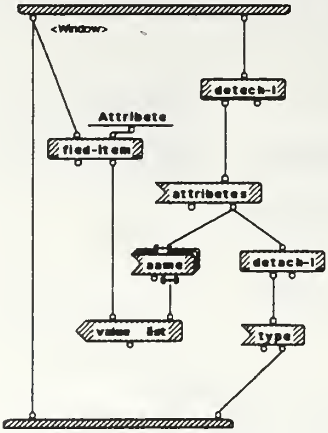
▨Design Form/read db 1:1get db name 1:1



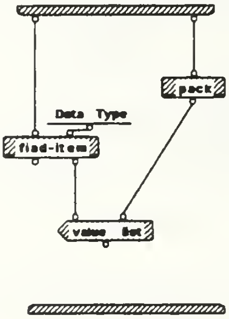
▨Design Form/read db 1:1set relation pop-up 1:1



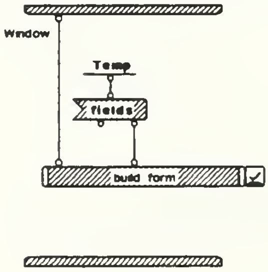
▨Design Form/read db 1:1get attributes 1:1



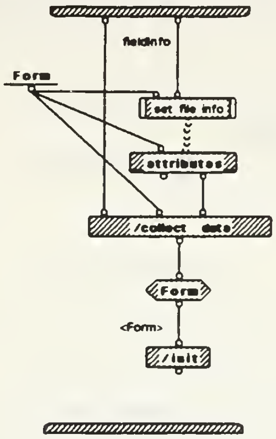
▨Design Form/read db 1:1get data type 1:1



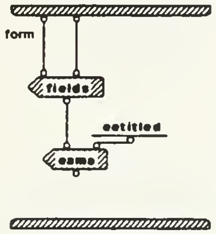
▨Design Form/build form 1:1



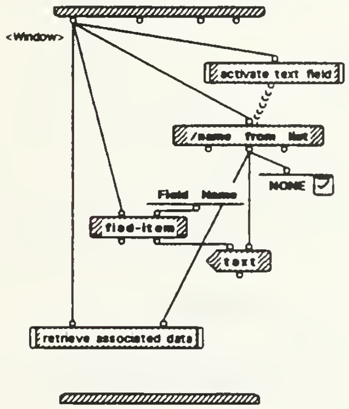
▨Design Form/build form 1:1build form 1:1



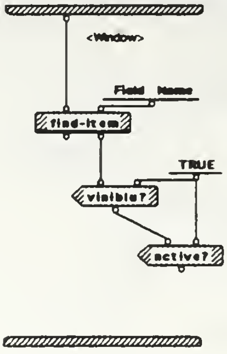
▨Design Form/build form 1:1build form 1:1set file info 1:1



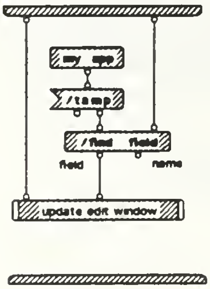
▨Design Form/get name 1:1



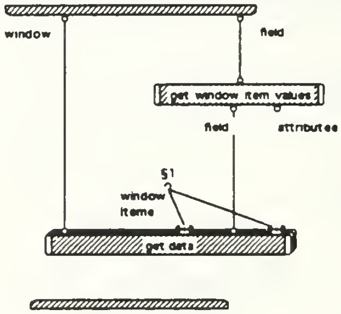
▨Design Form/get name 1:activate text field 1:1



▨Design Form/get name 1:retrieve associated data 1:1

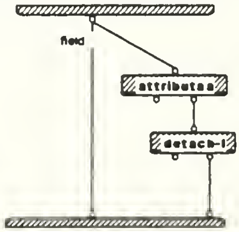


▨Design Form/get name 1:retrieve associated data 1:update edit window 1:1

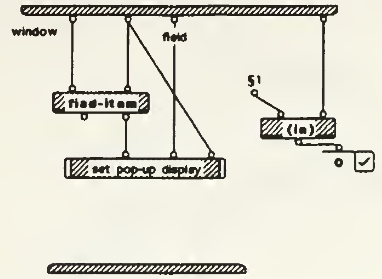


§1 ("Field Type" "Relation" "Attribute" "Data Type" "Font" "Font Size")

▨Design Form/get name 1:1retrieve associated data 1 1update edit window 1 1get window item values 1 1

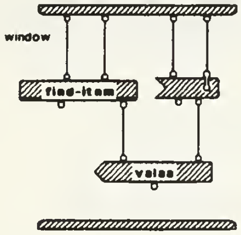


Design Form/get name 1:1retrieve associated data 1:1update edit window 1:1get data 1 2

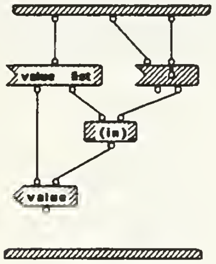


§1 ("Relation" "Attribute" "Date Type" "Font" "Font Size")

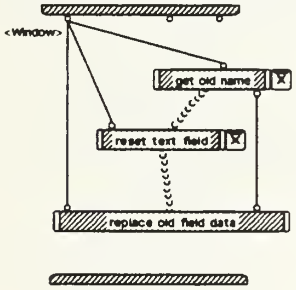
Design Form/get name 1:1retrieve associated data 1:1update edit window 1:1get data 2 2



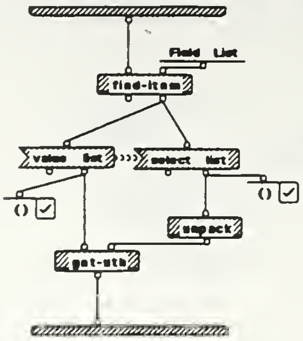
Design Form/get name 1:1retrieve associated data 1:1update edit window 1:1get data 1:2set pop-up display 1.1



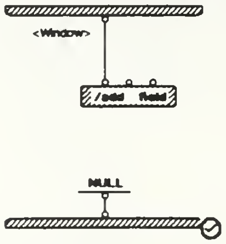
Design Form/okay edit field 1:1



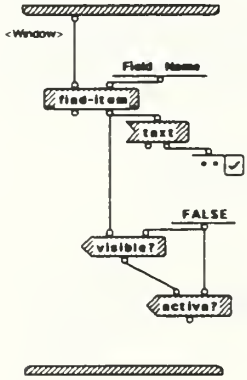
▨Design Form/okay edit field 1:1get old name 1:2



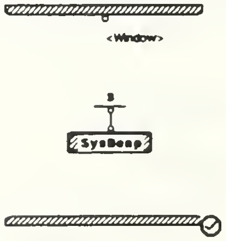
▨Design Form/okay edit field 1:1get old name 2:2



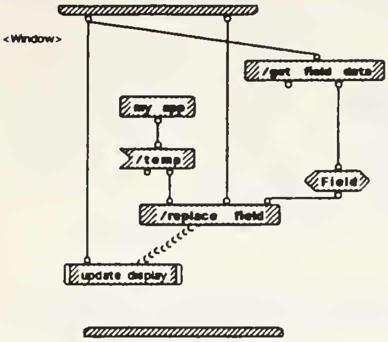
▨Design Form/okay edit field 1:1reset text field 1:2



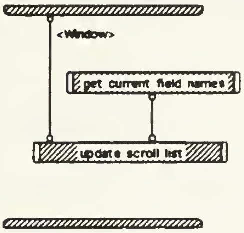
▨Design Form/okay edit field 1:1reset text field 2:2



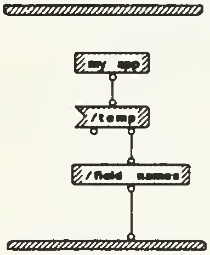
▨Design Form/okay edit field 1:1replace old field data 1:1



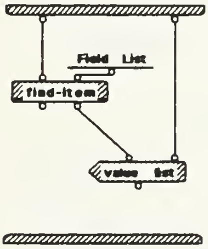
▨Design Form/okay edit field 1:1replace old field data 1:1update display 1:1



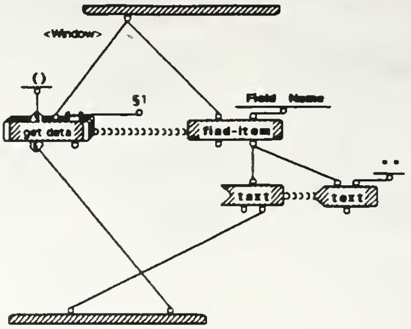
▨Design Form/okay edit field 1·1replace old field data 1:1update display 1·1get current field names 1 1



▨Design Form/okay edit field 1·1replace old field dete 1.1updete dieplay 1:1update ecroll list 1 1

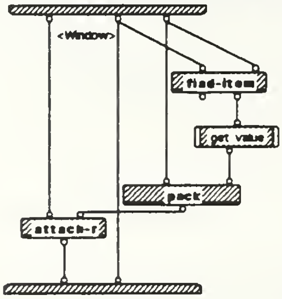


▨Design Form/get field data 1:1

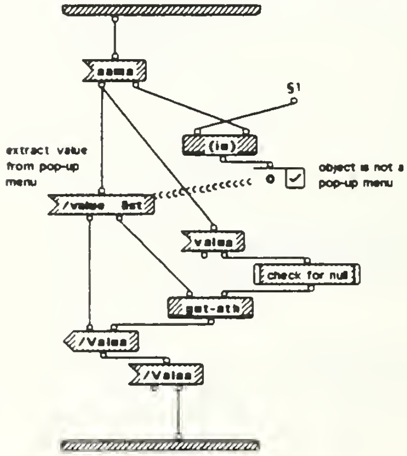


$1 ("Field Name" "Field Type" "Relation" "Attribute" "Data Type" "Font" "Font Size")

▨Design Form/get field data 1:1get data 1:1

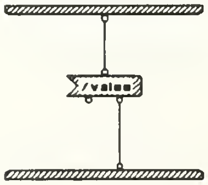


▨Design Form/get field data 1:1get data 1:1get value 1:2

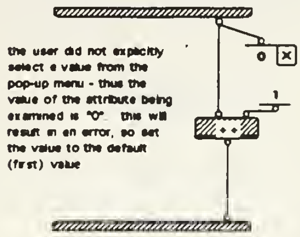


$1 ("Relation" "Attribute" "Data Type" "Font" "Font Size")

▨Design Form/get field data 1:1get data 1:1get value 2:2

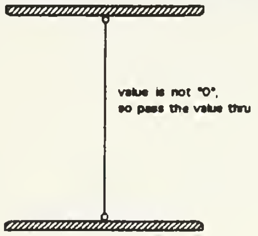


▨Design Form/get field data 1:1get data 1:1get value 1:2check for null 1:2



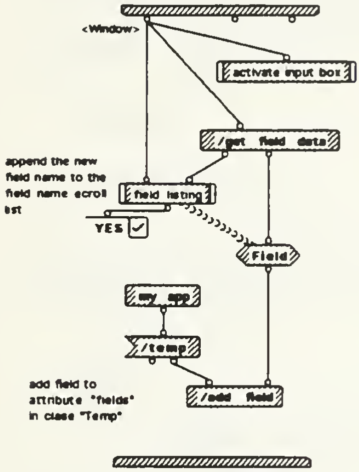
the user did not explicitly select e value from the pop-up menu - thus the value of the attribute being examined is "O". this will result in en error, so set the value to the default (first) value

▨Design Form/get field data 1:1get data 1:1get value 1:2check for null 2:2



value is not "O", so pass the value thru

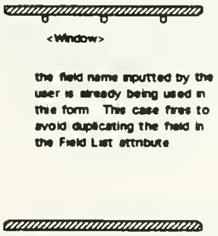
▨Design Form/add field 1:2



append the new field name to the field name ecroll list

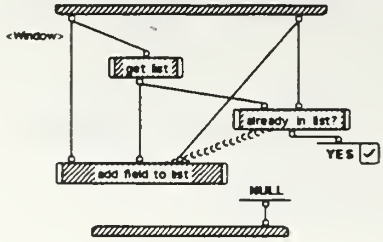
add field to attribute "fields" in clase "Temp"

▨Design Form/add field 2:2

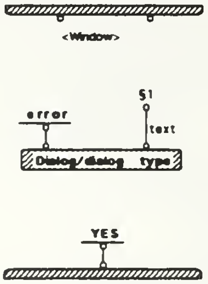


the field name inputted by the user is already being used in this form This case fires to avoid duplicating the field in the Field List attribute

▨Design Form/add field 1:2field listing 1:2

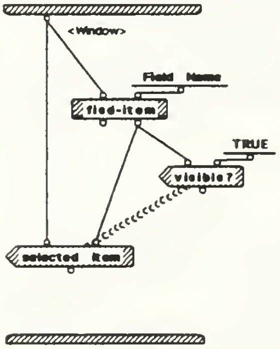


▨Design Form/add field 1:2field listing 2:2

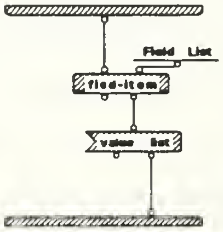


§1 That field name is already in use

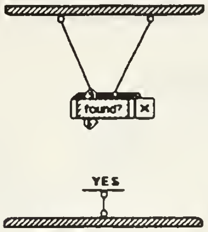
▨Design Form/add field 1:2activate input box 1:1



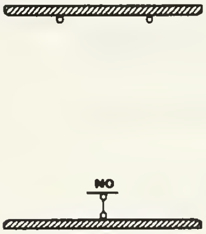
▨Design Form/add field 1:2field listing 1:2get list 1:1



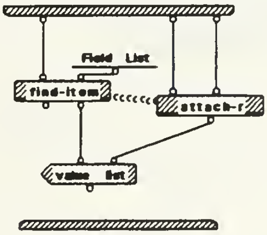
▨Design Form/add field 1:2field listing 1:2already in list? 1:2



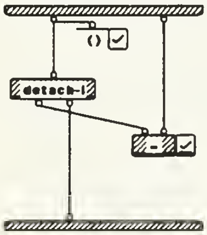
▨Design Form/add field 1:2field listing 1:2already in list? 2:2



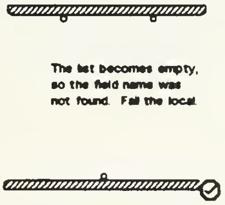
▨Design Form/add field 1:2field listing 1:2add field to list 1:1



▨Design Form/add field 1:2field listing 1:2already in list? 1:2found? 1:2

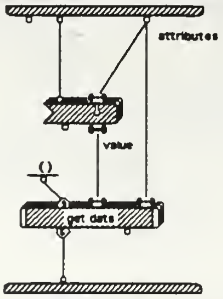


▨Design Form/add field 1:2field listing 1:2already in list? 1:2found? 2:2

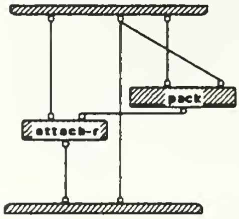


The list becomes empty, so the field name was not found. Fail the local.

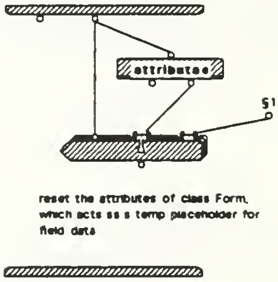
▨Design Form/collect data 1:1



▨Design Form/collect data 1:1get data 1:1



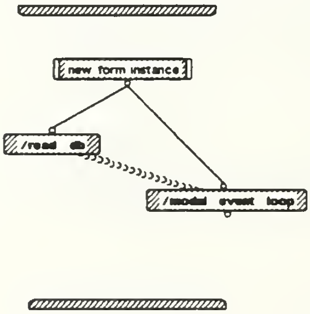
▨Design Form/reset form 1:1



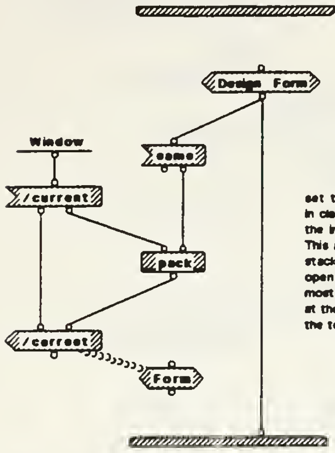
reset the attributes of class Form, which acts as a temp placeholder for field data

§1 ("" () "" FALSE)

▨Design Form/new form 1:1

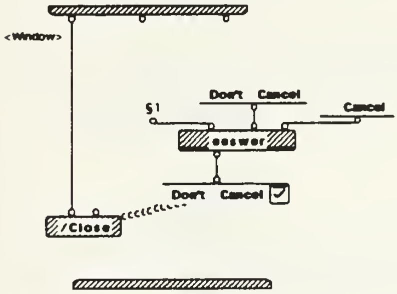


▨Design Form/new form 1:1new form instance 1:1



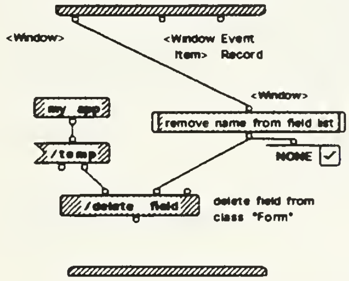
set the attribute "current" in class Window to reflect the instance of "New Form." This attribute acts like a stack, maintaining a list of open windows, with the left-most name in the list being at the top of the stack (ie - the topmost window)

▨Design Form/cancel dialog 1:1



§1 Cancel "Design Form" operation?

▨Design Form/delete name 1:2

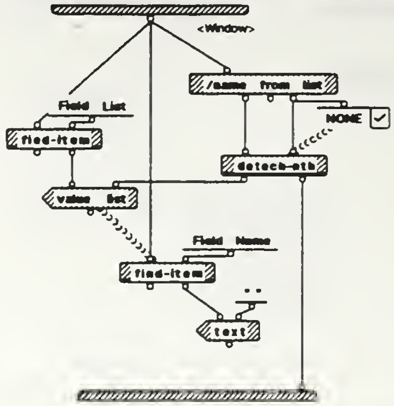


delete field from class "Form"

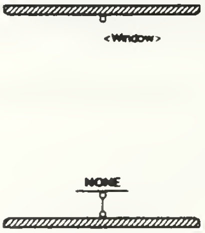
▨Design Form/delete name 2:2

<Window><Window Event
Item> Record

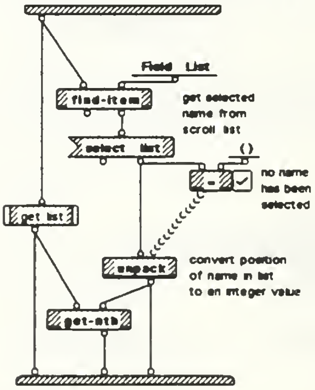
▨Design Form/delete name 1:2remove name from field list 1:2



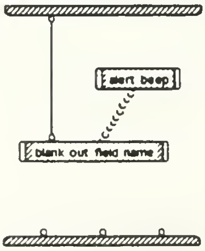
▨Design Form/delete name 1:2remove name from field list 2:2



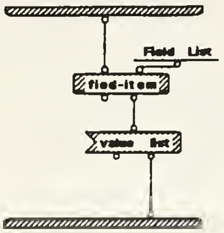
▨Design Form/name from list 1:2



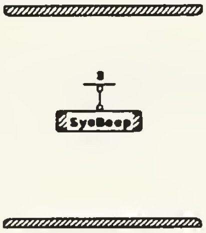
▨Design Form/name from list 2:2



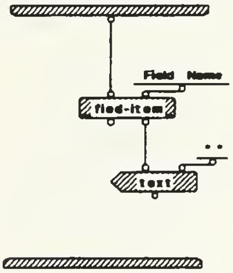
▨Design Form/name from list 1:2get list 1:1



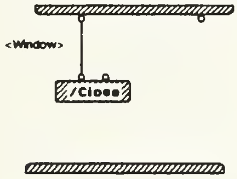
▨Design Form/name from list 2:2alert beep 1:1



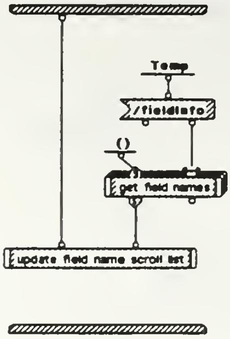
▨Design Form/name from list 2:2blank out field name 1:1



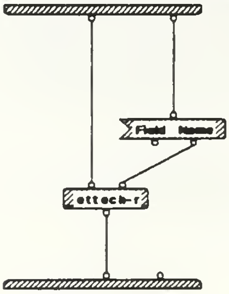
▨Design Form/close 1:1



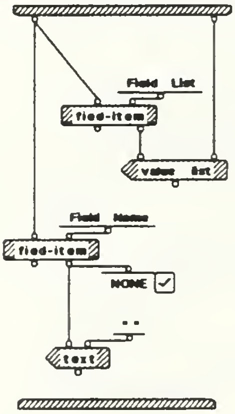
▨Design Form/update list 1:1



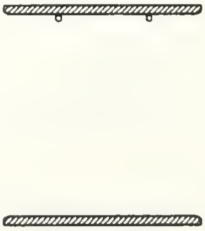
▨Design Form/update list 1:1get field names 1:1

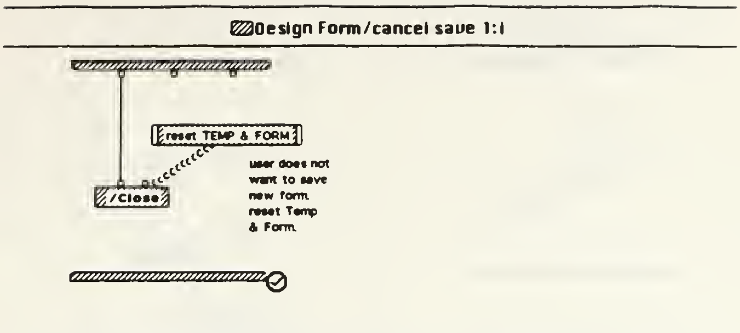


▨Design Form/update list 1:1update field name scroll list 1:2



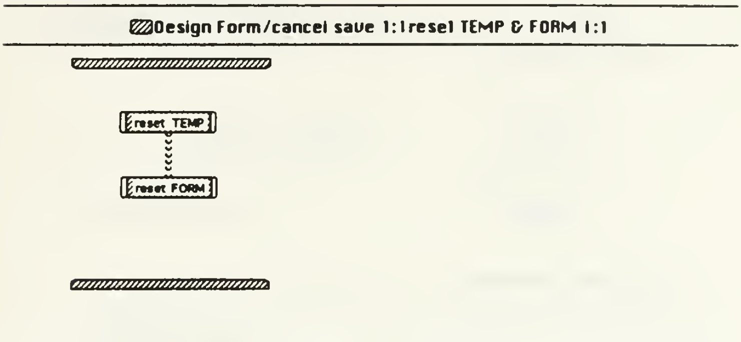
▨Design Form/update list 1:1update field name scroll list 2:2



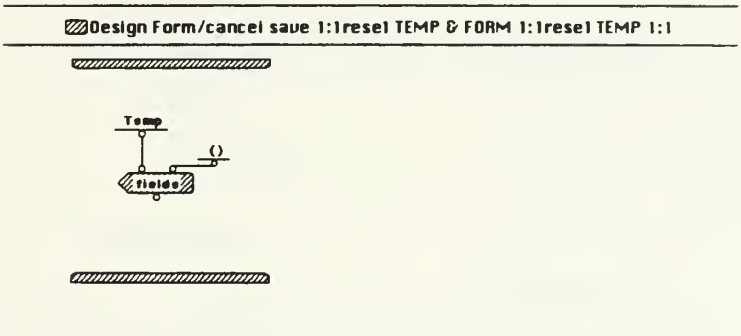


▨Design Form/cancel save 1:1

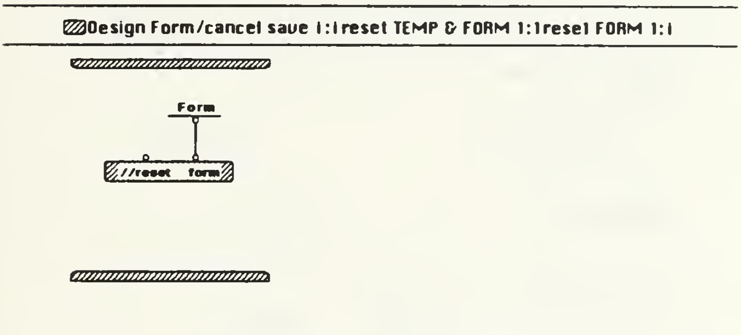
user does not want to save new form. reset Temp & Form.



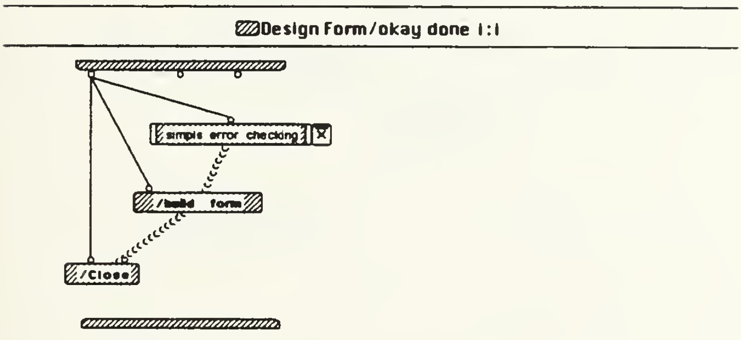
▨Design Form/cancel save 1:1reset TEMP & FORM 1:1



▨Design Form/cancel save 1:1reset TEMP & FORM 1:1reset TEMP 1:1

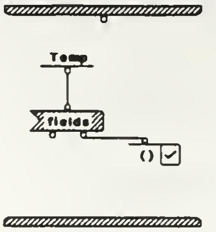


▨Design Form/cancel save 1:1reset TEMP & FORM 1:1reset FORM 1:1

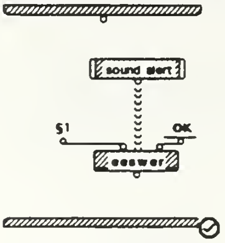


▨Design Form/okay done 1:1

▨Design Form/okay done 1:1simple error checking 1:2

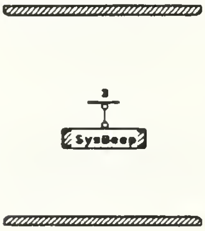


▨Design Form/okay done 1:1simple error checking 2:2

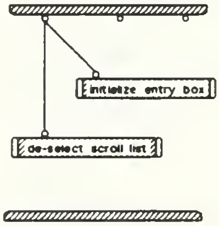


§1 No fields have been defined for this form.

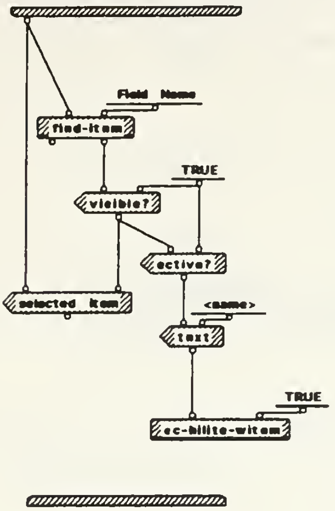
▨Design Form/okay done 1:1simple error checking 2:2sound alert 1:1



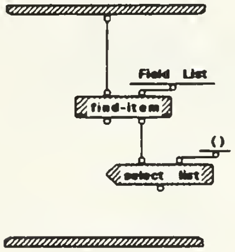
▨Design Form/new field input 1:1



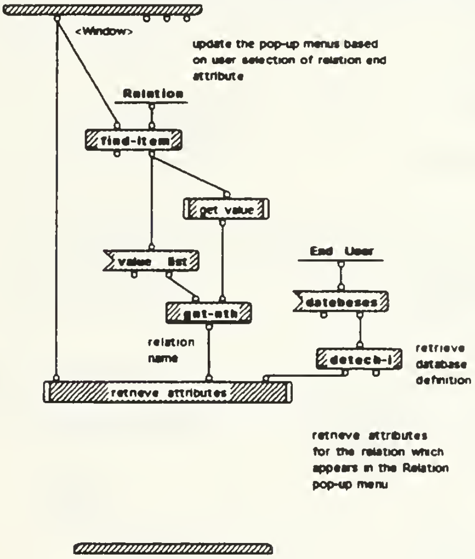
Design Form/new field input 1:1initialize entry box 1:1



Design Form/new field input 1:1 de-select scroll list 1:1



Design Form/set attribute pop-up 1:1

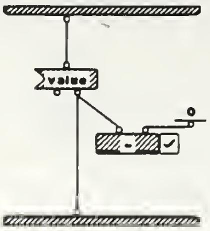


update the pop-up menus based on user selection of relation and attribute

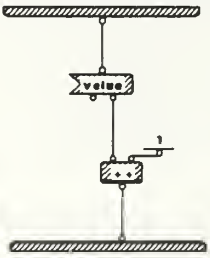
retrieve database definition

retrieve attributes for the relation which appears in the Relation pop-up menu

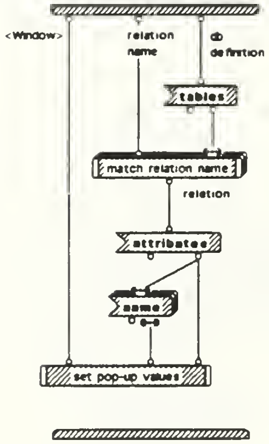
▨Design Form/set attribute pop-up 1:1get value 1:2



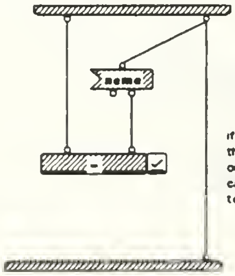
▨Design Form/set attribute pop-up 1:1get value 2:2



▨Design Form/set attribute pop-up 1:1retrieve attributes 1:1

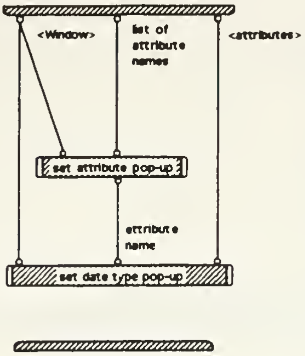


▨Design Form/set attribute pop-up 1 1retrieve attributes 1:1match relation name 1 1

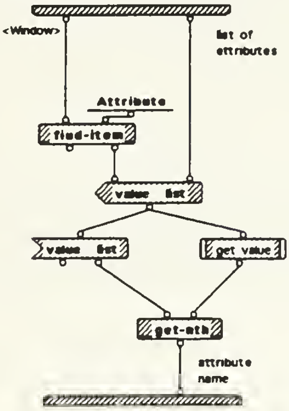


if a match is found, the name will be output on the case output bar terminal

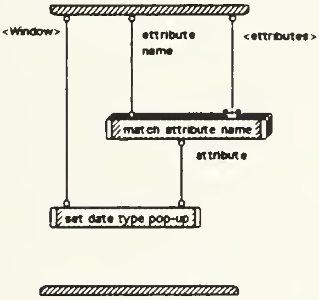
Design Form/set attribute pop-up 1:1retrieve attributes 1:1set pop-up values 1:1



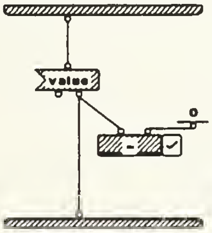
Design Form/set attribute pop-up 1:1retrieve attributes 1:1set pop-up values 1:1set attribute pop-up 1:1



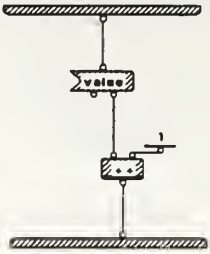
Design Form/set attribute pop-up 1:1retrieve attributes 1:1set pop-up values 1:1set data type pop-up 1:1



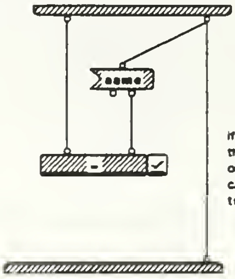
Design Form/set attribute pop-up 1:1retrieve attributes 1:1set pop-up values 1:1set attribute pop-up 1:1get valu
1:2



▨Design Form/set attribute pop-up 1:1retrieve attributes 1 1set pop-up values 1:1set attribute pop-up 1·1get valu
▨ 2.2

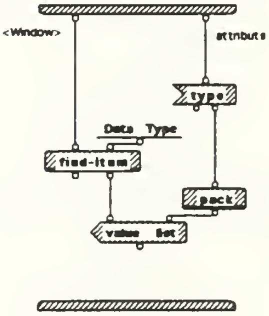


▨Design Form/set attribute pop-up 1 1retrieve attributes 1.1set pop-up values 1.1set date type pop-up 1 1match
▨tribute name 1.1



If a match is found, the name will be output on the case output bar terminal

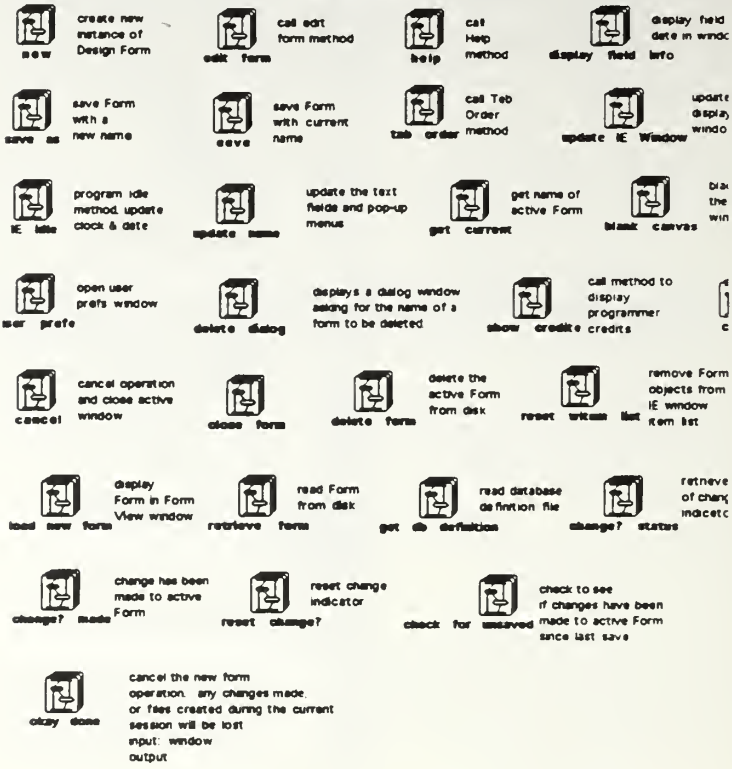
▨Design Form/set attribute pop-up 1 1retrieve attributes 1.1set pop-up values 1 1set data type pop-up 1 1set dat
▨type pop-up 1:1



▽ IE Window

```
      • •
       ⬡
    current
-------------------------------
  "Interface E
       ▽
      name

      NULL
       ▽
     owner

     FALSE
       ▽
    active?

      NULL
       ▽
 window  record

       4
       ▽
     def ID

      TRUE
       ▽
     modal?

      TRUE
       ▽
     close?

      NULL
       ▽
 selected  item

    | 40 3 |
       ▽
    location

  | 427 622 |
       ▽
      size

      • •
       ▽
activate  method

   "/close"
       ▽
 close  method

  "/IE  idle"
       ▽
  idle  method

      • •
       ▽
  key  method

 ( <<Text>> <.
       ▼
   item  list

      NULL      <Database> associated
       ▽        to this window
    database

     FALSE
       ▽
    changes?
```

IE Window



new
create new instance of Design Form

edit form
call edit form method

help
call Help method

display field info
display field data in windc

save as
save Form with a new name

save
save Form with current name

tab order
call Tab Order method

update IE Window
update display windo

IE idle
program idle method. update clock & date

update name
update the text fields and pop-up menus

get current
get name of active Form

blank canvas
bla the win

user prefs
open user prefs window

delete dialog
displays a dialog window asking for the name of a form to be deleted

show credits
call method to display programmer credits

c

cancel
cancel operation and close active window

close form

delete form
delete the active Form from disk

reset witem list
remove Form objects from IE window item list

load new form
display Form in Form View window

retrieve form
read Form from disk

get db definition
read database definition file

change? status
retrieve of chang indicatc

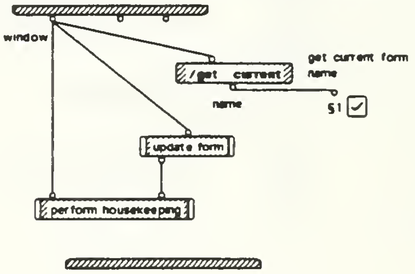
change? made
change has been made to active Form

reset change?
reset change indicator

check for unsaved
check to see if changes have been made to active Form since last save

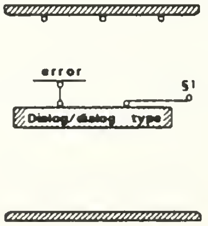
okay done
cancel the new form operation. any changes made, or files created during the current session will be lost
input: window
output

IE Window/save as 1:2



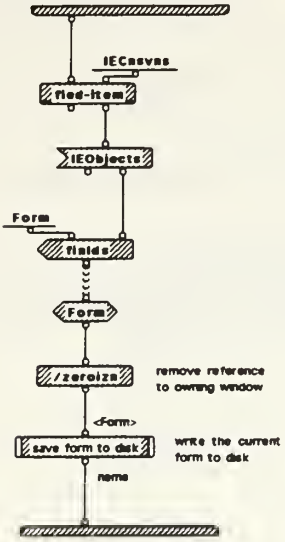
§1 <no currently active form>

IE Window/save as 2:2



§1 You can only save the currently active form. There is no active form to save

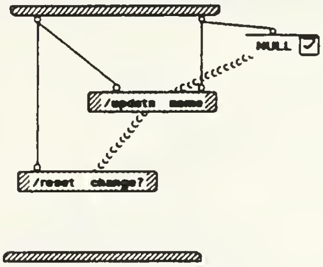
▨IE Window/save as 1:2update form 1:1



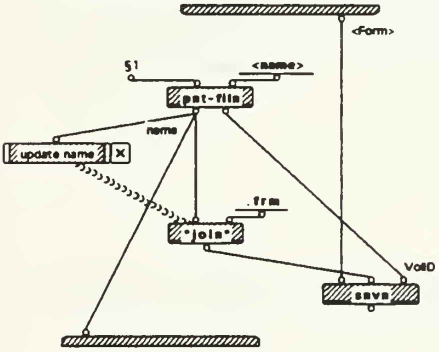
remove reference to owning window

write the current form to disk

▨IE Window/save as 1:2perform housekeeping 1:1

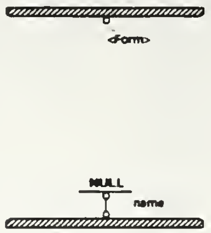


▨IE Window/save as 1:2update form 1:1save form to disk 1:2

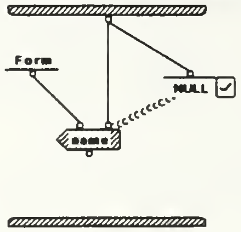


§1 Save Form As

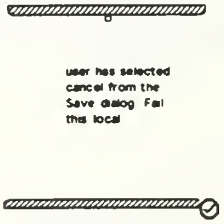
▨IE Window/save as 1:2update form 1:1save form to disk 2:2



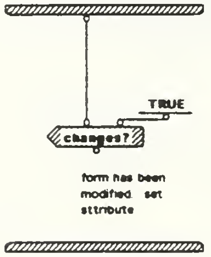
▨IE Window/save as 1:2update form 1:1save form to disk 1:2update name 1:2



▨IE Window/save as 1:2update form 1:1save form to disk 1:2update name 2:2

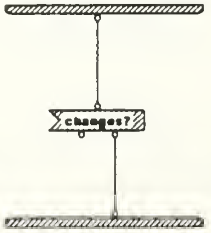

user has selected cancel from the Save dialog. Fail this local

▨IE Window/change? made 1:1

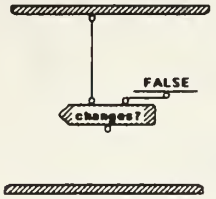


form has been modified, set attribute

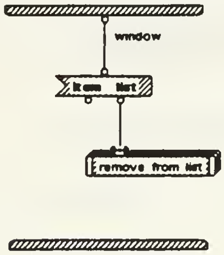
▨IE Window/change? status 1:1



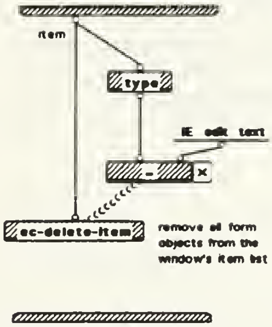
▨IE Window/reset change? 1:1



▨IE Window/reset witem list 1:1

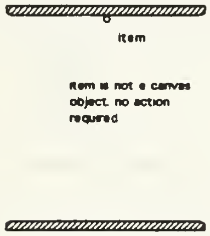


▨IE Window/reset witem list 1:1remove from list 1:2



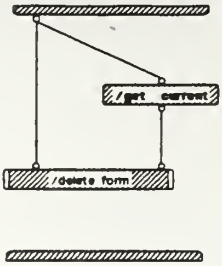
remove all form objects from the window's item list

▨IE Window/reset witem list 1:1remove from list 2:2

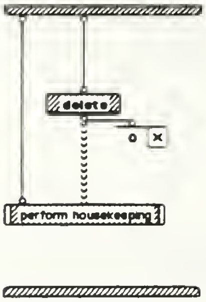


item is not a canvas object. no action required

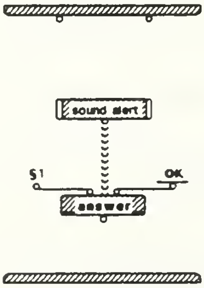
▨IE Window/delete form 1:1



▨IE Window/delete form 1:1/delete form 1:2

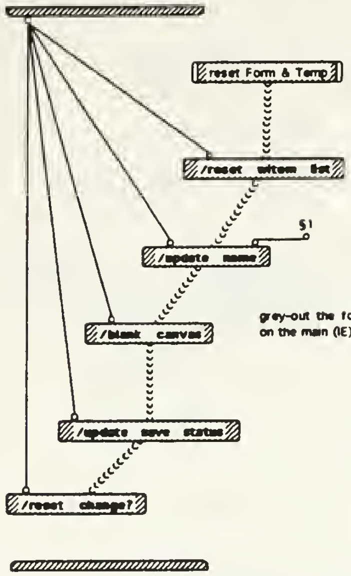


▨IE Window/delete form 1:1/delete form 2:2



§1. File error Unable to delete specified file Please verify name and location.

▨IE Window/delete form 1:1/delete form 1:2perform housekeeping 1:1



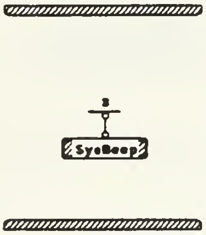
reset the attributes
of classes Form & Temp
This must be done before
reseting the attributes
of class Librarian

remove canvas objects
from the owner window's
item list

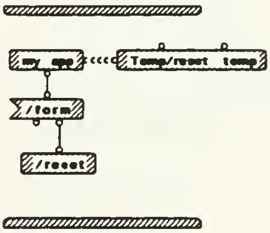
grey-out the form view canvas
on the main (IE) Window

§1. <no currently active form>

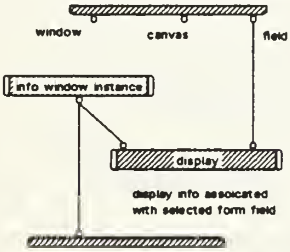
▨IE Window/delete form 1:1/delete form 2:2sound alert 1:1



▨IE Window/delete form 1:1/delete form 1:2perform housekeeping 1 1reset Form & Temp 1:1

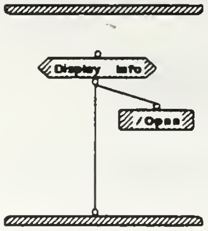


▨IE Window/display field info 1:1

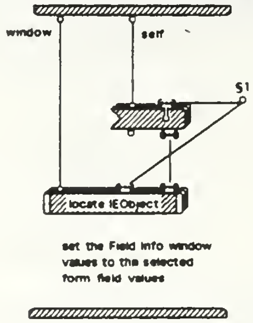


display info assoicated
with selected form field

IE Window/display field info 1:1info window instance 1:1



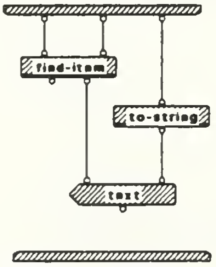
IE Window/display field info 1:1display 1:1



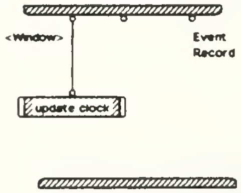
set the Field Info window
values to the selected
form field values

S1 ("Field Name" "Data Type" "Relation" "Attribute" "Field Type" "Font" "Font Size")

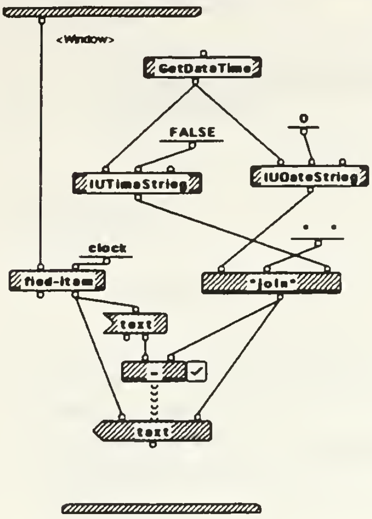
IE Window/display field info 1:1display 1:1locate IEObject 1:1



IE Window/IE idle 1:1



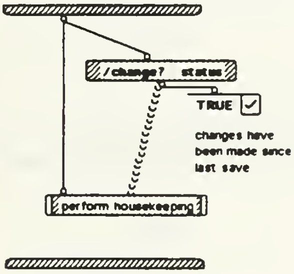
▨IE Window/IE Idle 1:1update clock 1:2



▨IE Window/IE Idle 1:1update clock 2:2

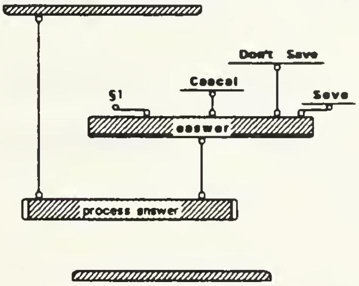
the time&date displayed in the window does not need to be updated no action required.

▨IE Window/check for unsaved changes 1:3



changes have been made since last save

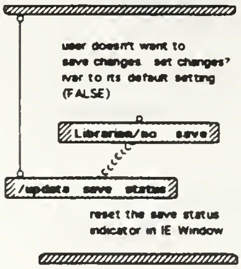
▨IE Window/check for unsaved changes 2:3



▨IE Window/check for unsaved changes 2:3

§1. Save changes before closing?

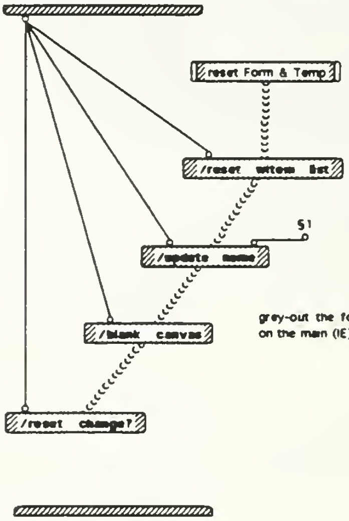
▨IE Window/check for unsaved changes 3:3



user doesn't want to
save changes. set changes?
var to its default setting
(FALSE)

reset the save status
indicator in IE Window

▨IE Window/check for unsaved changes 1:3perform housekeeping 1:1



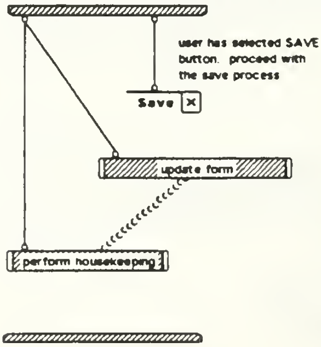
reset the attributes
of classes Form & Temp
This must be done before
reseting the attributes
of class Librarian

remove canvss objects
from the owner window's
item list

grey-out the form view canvas
on the main (IE) Window

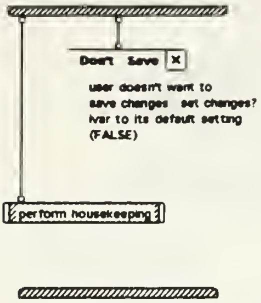
§1 <no currently active form>

▨IE Window/check for unsaved changes 2:3process answer 1:3



user has selected SAVE
button. proceed with
the save process

▨IE Window/check for unsaved changes 2:3process answer 2:3

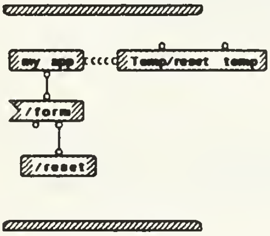


user doesn't want to
save changes set changes?
Ivar to its default setting
(FALSE)

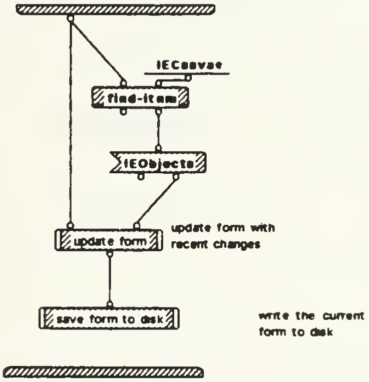
▨IE Window/check for unsaved changes 2:3process answer 3:3

user has selected the
CANCEL button
fail this local do
not proceed with
the save or the
close operation

▨IE Window/check for unsaved changes 1:3perform housekeeping 1 1reset Form & Temp 1 1



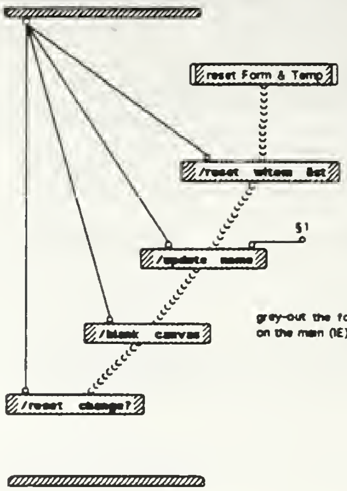
▨IE Window/check for unsaved changes 2:3process answer 1:3update form 1:1



update form with
recent changes

write the current
form to disk

▨IE Window/check for unsaved changes 2:3process answer 1:3perform housekeeping 1:1



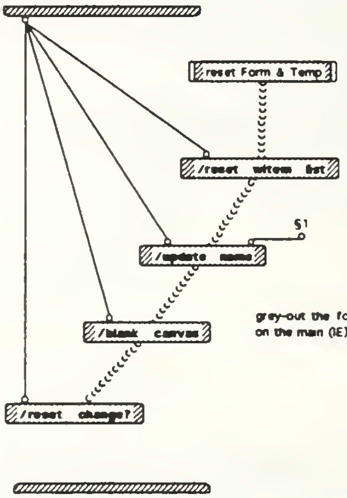
reset the attributes
of classes Form & Temp
This must be done before
resetting the attributes
of class Librarian

remove canvas objects
from the owner window's
item list

grey-out the form view canvas
on the main (IE) Window

§1 <no currently active form>

▨IE Window/check for unsaved changes 2:3process answer 2:3perform housekeeping 1:1



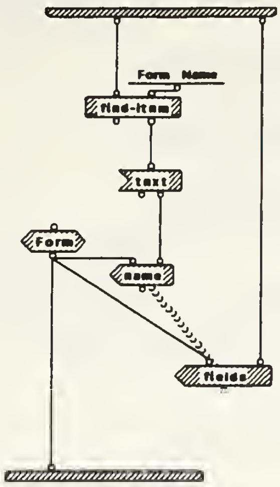
reset the attributes
of classes Form & Temp.
This must be done before
resetting the attributes
of class Librarian.

remove canvas objects
from the owner window's
item list

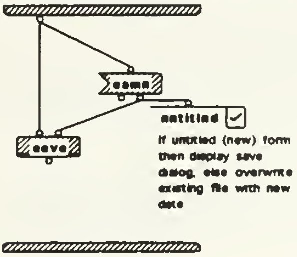
grey-out the form view canvas
on the main (IE) Window

§1 <no currently active form>

▨IE Window/check for unsaved changes 2:3process answer 1:3update form 1:1update form 1 1

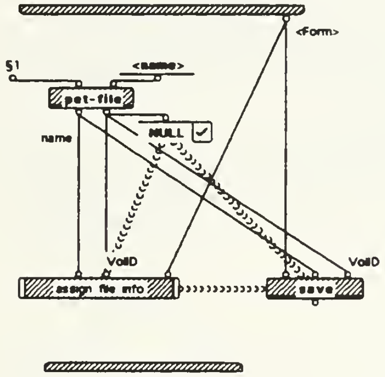


▨IE Window/check for unsaved changes 2:3process answer 1:3update form 1.1save form to disk 1:3



If untitled (new) form then display save dialog, else overwrite existing file with new date

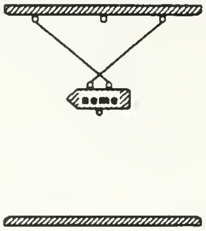
▨IE Window/check for unsaved changes 2:3process answer 1:3update form 1 1save form to disk 2.3



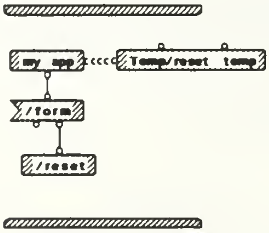
§1. Save Form As

▨IE Window/check for unsaved changes 2:3process answer 1:2update form 1.1save form to disk 3:3

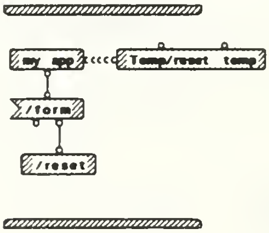
▨IE Window/check for unsaved changes 2:3process answer 1 3update form 1 1save form to disk 2 3assign file info 1
▨ 1



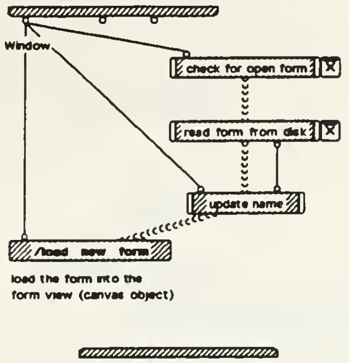
▨IE Window/check for unsaved changes 2 3process answer 1:3perform housekeeping 1 1reset Form & Temp 1.1



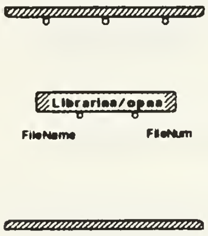
▨IE Window/check for unsaved changes 2.3process answer 2 3perform housekeeping 1 1reset Form & Temp 1.1



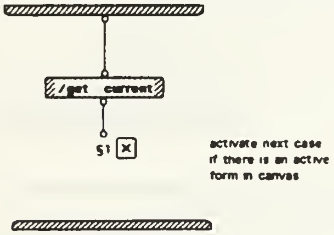
▨IE Window/retrieve form 1:2



▨IE Window/retrieve form 2:2

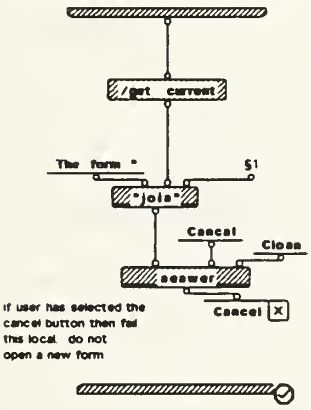


▨IE Window/retrieve form 1:2check for open form 1:3



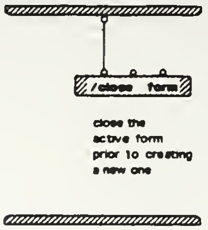
§1 <no currently active form>

▨IE Window/retrieve form 1:2check for open form 2:3

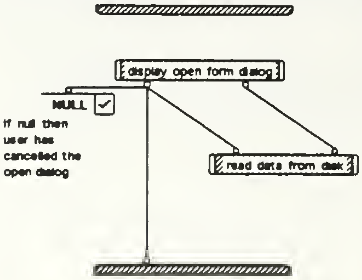

/get current
The form "
§1
"join"
Cancel
Close
answer
Cancel ☒
if user has selected the cancel button then fail this local. do not open a new form

§1 " must be closed before opening a new form. Close the form?

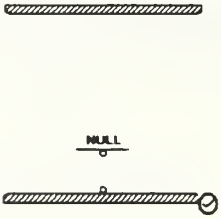
▨IE Window/retrieve form 1:2check for open form 3:3



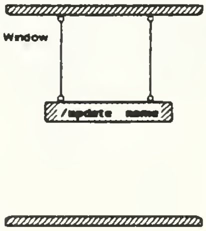
▨IE Window/retrieve form 1:2read form from disk 1:2



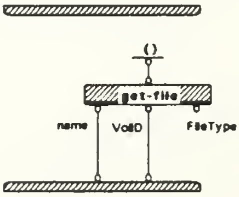
▨IE Window/retrieve form 1:2read form from disk 2:2



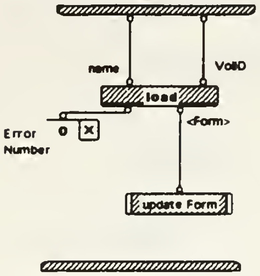
▨IE Window/retrieve form 1:2update name 1:1



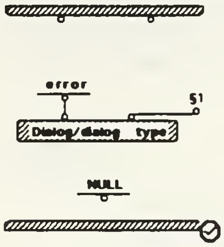
▨IE Window/retrieve form 1 2read form from disk 1.2display open form dialog 1 1



▨IE Window/retrieve form 1:2read form from disk 1:2read data from disk 1:2

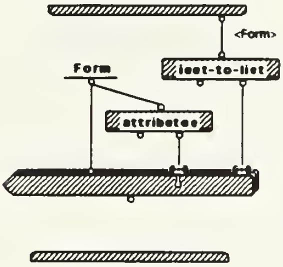


▨IE Window/retrieve form 1:2read form from disk 1:2read data from disk 2:2

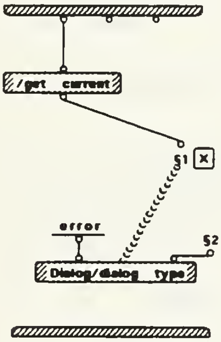


§1. Unable to read file from disk

▨IE Window/retrieve form 1:2read form from disk 1:2read data from disk 1:2update Form 1:1

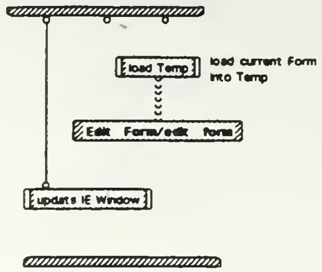


▨IE Window/edit form 1:2

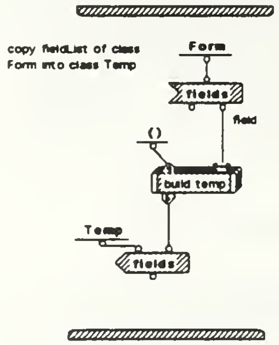


§1 <no currently active form>
§2 You can only edit the currently active form. There is no active form to edit

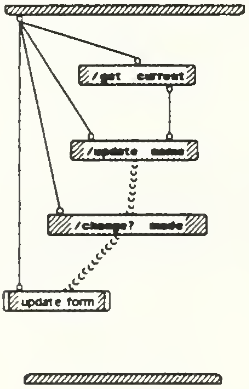
▨IE Window/edit form 2:2



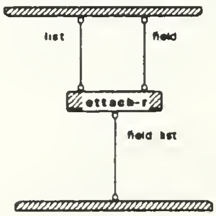
▨IE Window/edit form 2:2load Temp 1:1



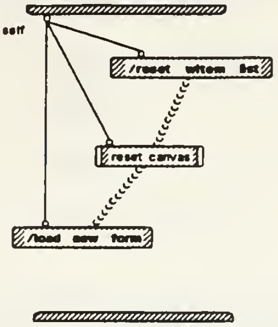
▨IE Window/edit form 2:2update IE Window 1:1



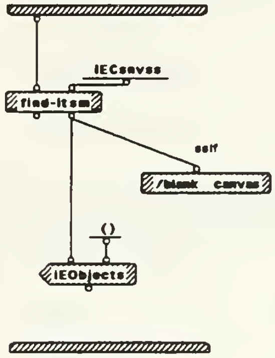
▨IE Window/edit form 2:2load Temp 1:1build temp 1:1



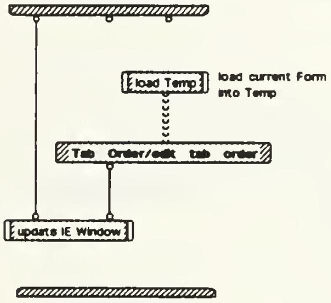
▨IE Window/edit form 2:2update IE Window 1:1update form 1:1



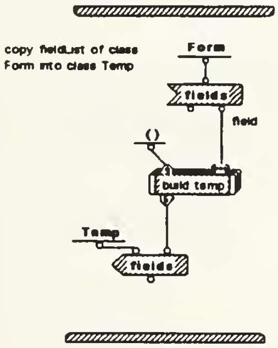
▨IE Window/edit form 2:2update IE Window 1:1update form 1:1reset canvas 1:1



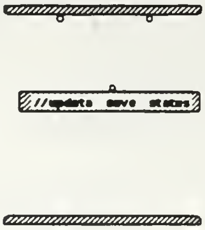
▨IE Window/tab order 1:1



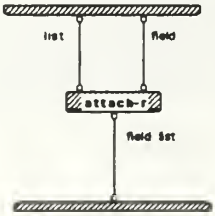
▨IE Window/tab order 1:1load Temp 1:1



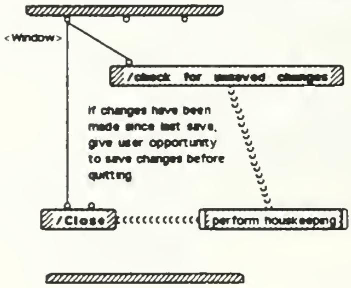
▨IE Window/tab order 1:1update IE Window 1:1



▨IE Window/tab order 1:1load Temp 1:1build temp 1:1

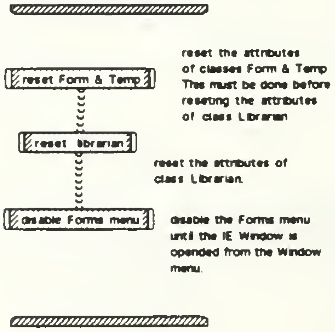


▨IE Window/okay done 1:1



If changes have been made since last save, give user opportunity to save changes before quitting

▨IE Window/okay done 1:1perform houskeeping 1:1

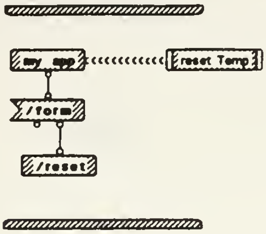


reset the attributes of classes Form & Temp This must be done before reseting the attributes of class Librarian

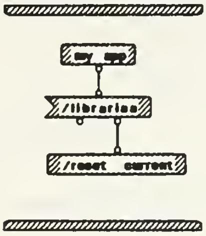
reset the attributes of class Librarian.

disable the Forms menu until the IE Window is opended from the Window menu.

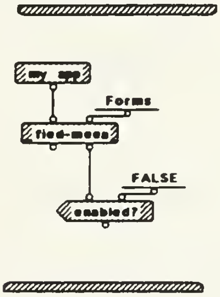
▨IE Window/okay done 1:1perform houskeeping 1:1reset Form & Temp 1:1



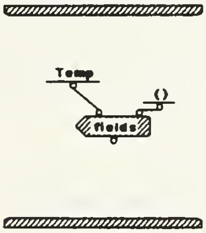
▨IE Window/okay done 1:1perform houskeeping 1:1reset librarian 1:1



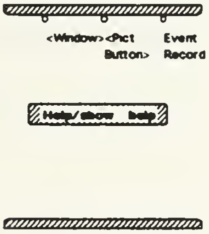
▨IE Window/okay done 1:1perform houskeeping 1:1disable Forms menu 1:1



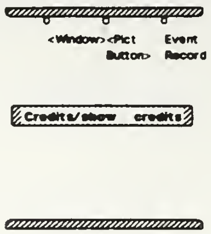
▨IE Window/okay done 1:1perform houskeeping 1:1reset Form & Temp 1:1reset Temp 1:1



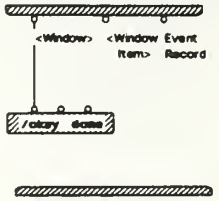
▨IE Window/help 1:1



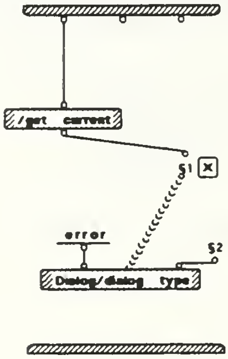
▨IE Window/show credits 1:1



▨IE Window/cancel 1:1

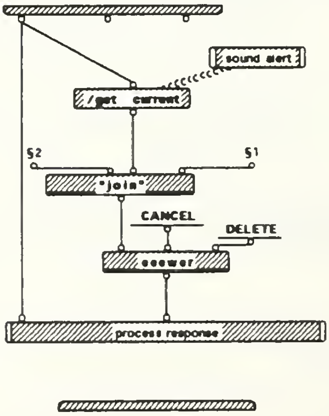


▨IE Window/delete dialog 1:2



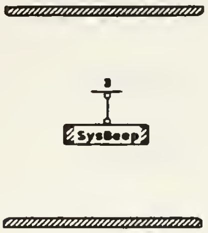
§1 <no currently active form>
§2 You can only delete the currently active form. There is no active form to delete

▨IE Window/delete dialog 2:2

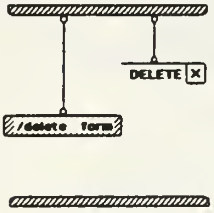


§1 " This operation can't be undone
§2 You are about to delete the form "

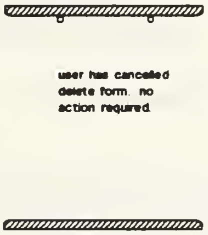
▨IE Window/delete dialog 2:2sound alerl 1:1



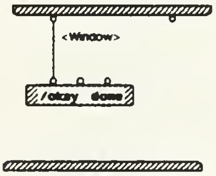
▨IE Window/delele dialog 2:2process response 1:2



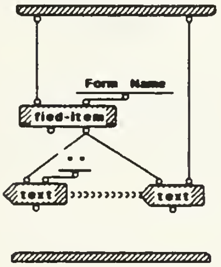
▨IE Window/delele dialog 2:2process response 2:2


user has cancelled delete form. no action required

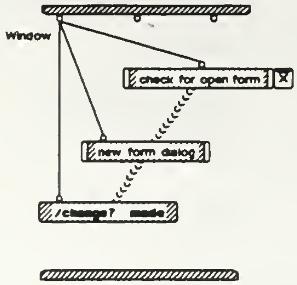
▨IE Window/close 1:1



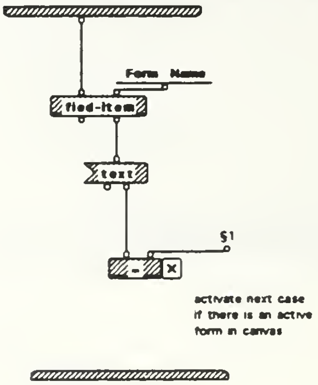
▨IE Window/update name 1:1



▨IE Window/new 1:1

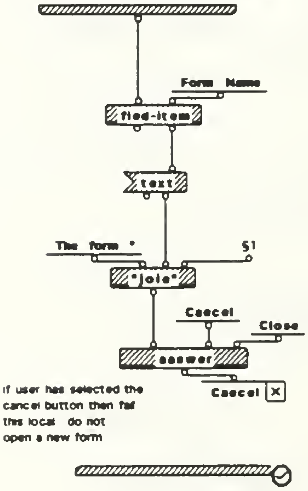


▨IE Window/new 1:1check for open form 1:3



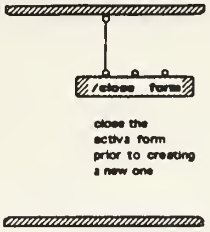
§1 <no currently active form>

▨IE Window/new 1:1check for open form 2:3

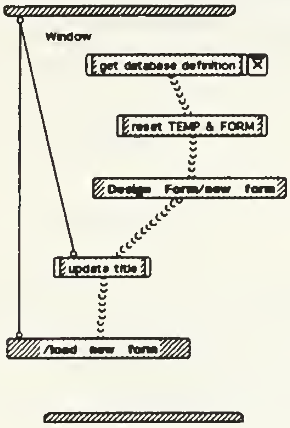


§1 " must be closed before opening a new form. Close the form?

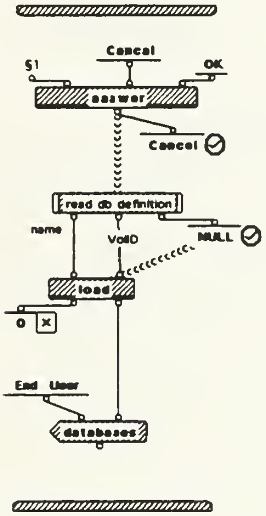
▨IE Window/new 1:1check for open form 3:3



▨IE Window/new 1:1new form dialog 1:1

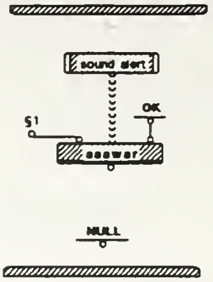


▨IE Window/new 1:1new form dialog 1:1get database definition 1:2



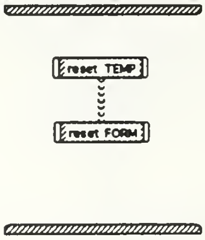
§1 Creating a new Form requires opening the database definition file OK to open the file?

▨IE Window/new 1:1new form dialog 1:1get database definition 2:2

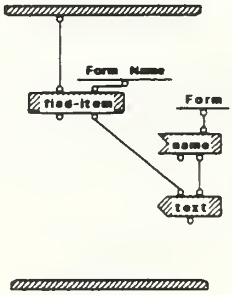


§1 Unable to load database definition file. Please verify name & location.

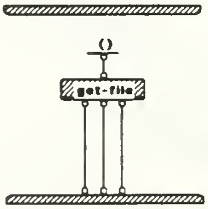
▨IE Window/new 1:1new form dialog 1:1reset TEMP & FORM 1:1



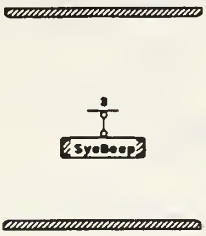
▨IE Window/new 1:1new form dialog 1:1update title 1:1



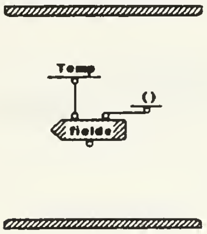
▨IE Window/new 1:1new form dialog 1 1get database definition 1 2read db definition 1 1



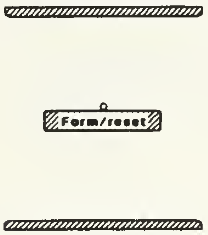
▨IE Window/new 1:1new form dialog 1:1get database definition 2:2sound alert 1:1



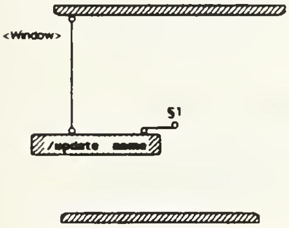
▨IE Window/new 1:1new form dialog 1:1reset TEMP & FORM 1:1reset TEMP 1:1



▨IE Window/new 1:1new form dialog 1:1reset TEMP & FORM 1:1reset FORM 1:1

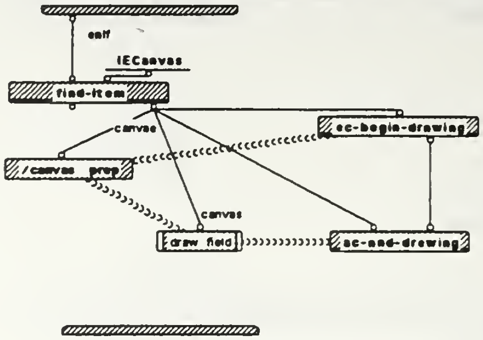


▨IE Window/update IE Window 1:1

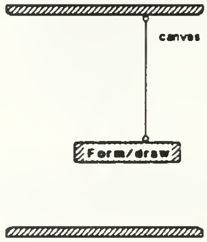


§1 <no currently active form>

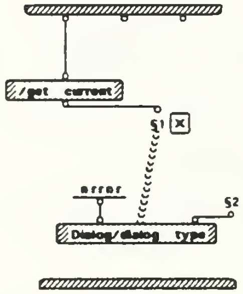
▨IE Window/load new form 1:1



▨IE Window/load new form 1:1draw field 1:1

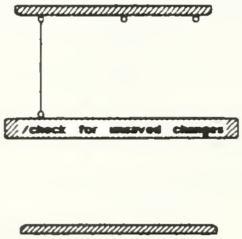


▨IE Window/close form 1:2

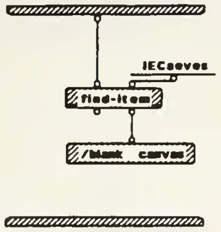


§1 <no currently active form>
§2 You can only close the currently active form. There is no active form to close

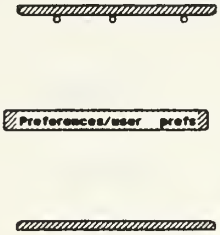
▨IE Window/close form 2:2



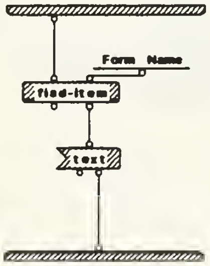
▨IE Window/blank canvas 1:1



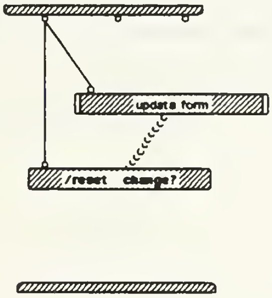
▨IE Window/user prefs 1:1



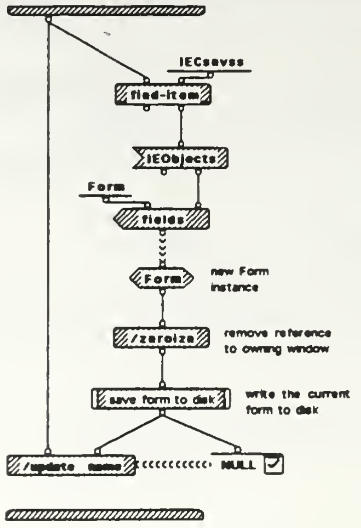
▨IE Window/get current 1:1



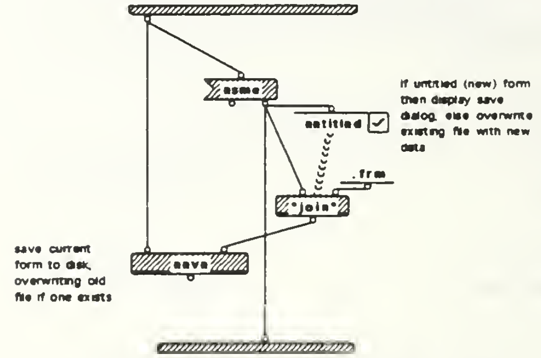
▨IE Window/save 1:1



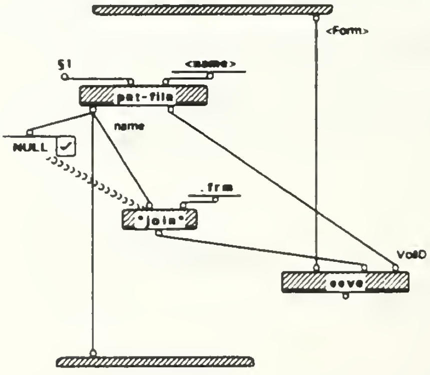
▨IE Window/save 1:1update form 1:1



▨IE Window/save 1:1update form 1:1save form to disk 1:3

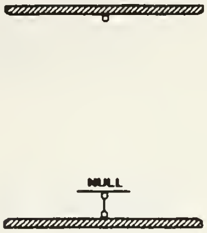


▨IE Window/save 1:1update form 1:1save form to disk 2:3

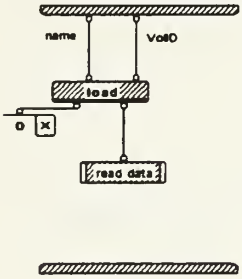


§1 Save Form As

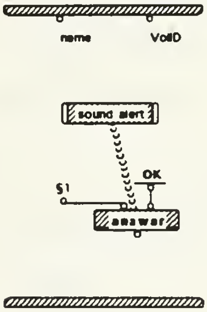
▨IE Window/save 1:1update form 1:1save form to disk 3:3



▨IE Window/get db definition 1:2

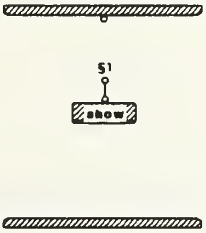


▨IE Window/get db definition 2:2



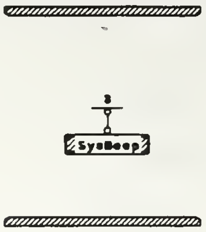
§1 Unable to load database definition file. Please confirm file name and location

▨IE Window/get db definition 1:2read data 1:1



§1 reading db definition

▨IE Window/get db definition 2:2sound alert 1:1



∇ Help

NULL
current

"Help Window
name

NULL
owner

FALSE
active?

NULL
window record

5
def ID

TRUE
modal?

FALSE
close?

NULL
selected item

(126 55)
location

(291 493)
size

• •
activate method

"/Close"
close method

• •
idle method

• •
key method

(<<Button>>_
item list

Help

helpClick detect mouse click in help window topic list

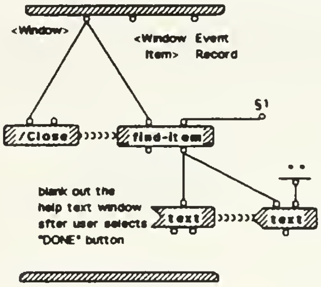
Help called from icon button. calls "show help"

help info display help text.
input: window; window item; order number of help topic in list.

show help open the help window & display default text describing how to use help

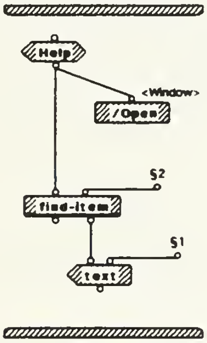
okay done close help window and blank out help scroll text.

Help/okay done 1:1


<Window>
<Window Item>
Event Record
§1
/Close
find-item
blank out the help text window after user selects "DONE" button
text
text
""

§1 Help Scroll Text

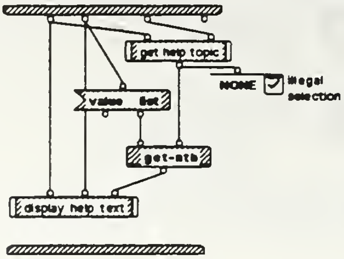
Help/show help 1:1



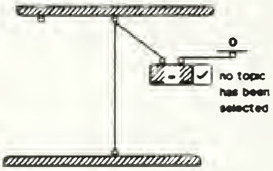
§1 Select a help topic by clicking on a subject heading in the scroll list. Text will appear in this window. dismiss the help window by selecting the DONE button

§2 Help Scroll Text

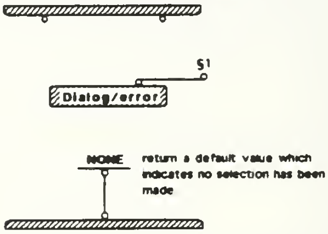
▨Help/help info 1:1



▨Help/help info 1:1get help topic 1:2

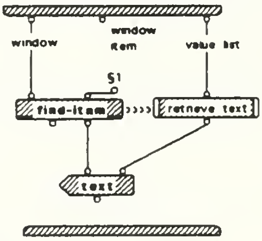


▨Help/help info 1:1get help topic 2:2


§1
Dialog/error
NONE
return a default value which indicates no selection has been made

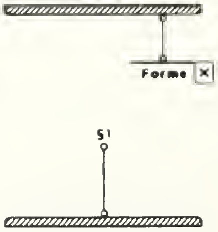
§1 Select a help topic by clicking a name in the Help Topics list, or choose DONE to close the window

▨Help/help info 1:1display help text 1:1



§1 Help Scroll Text

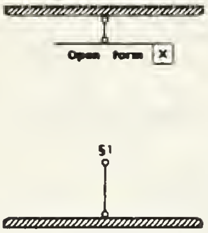
▨Help/help info 1:1display help text 1:1retrieve text 1:3



▨Help/help info 1:1display help text 1:1retrieve text 1:3

§1 A form is an object which enables a user to easily input data into the Amadeus Query Window Forms are created by selecting the "Create Form" icon from the icon bar, or by selecting "New" from the Forms menu in the menu bar Forms are stored on disk, and can be retrieved for use, editing or deleting, as necessary

▨Help/help info 1:1display help text 1:1retrieve text 2:3



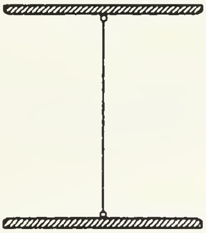
§1 Forms are stored on disk. Valid Interface Editor forms can be identified by the suffix ".frm" This suffix is added by the system automatically However, if you change the name of a form using the Macintosh finder (clicking on a form icon and highlighting the name, then typing a new name), IT IS IMPORTANT THAT YOU ADD THE SUFFIX .frm YOURSELF Amadeus will only recognize forms that end in this suffix

OPENING a form is accomplished in one of two ways

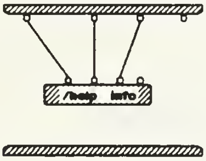
(1) select Open from the file menu, or
(2) select the Open Form icon (the second icon from the left in the icon bar)

A Macintosh hierarchical file dialog window will open, allowing you to select the form file to open. You can access all connected volumes (disk drives) from this window Select the file to open by highlighting the file name and pressing the OPEN button, or you can double-click on the file name. Choose the CANCEL button if you wish to dismiss the dialog windo without opening a form file

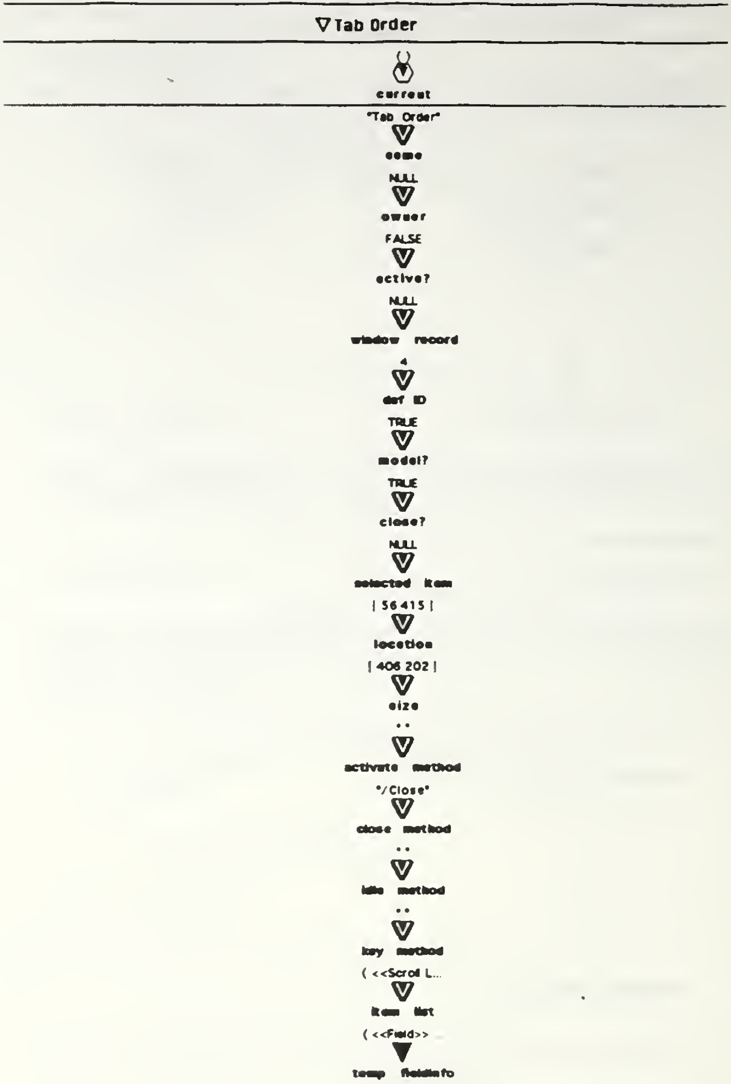
▨Help/help info 1:1display help text 1:1retrieve text 3:3



▨Help/helpClick 1:1



▨Help/Help 1:1



∇ Tab Order

current

"Tab Order"
name

NULL
owner

FALSE
active?

NULL
window record

4
def ID

TRUE
modal?

TRUE
close?

NULL
selected item

| 56 415 |
location

| 406 202 |
size

• •
activate method

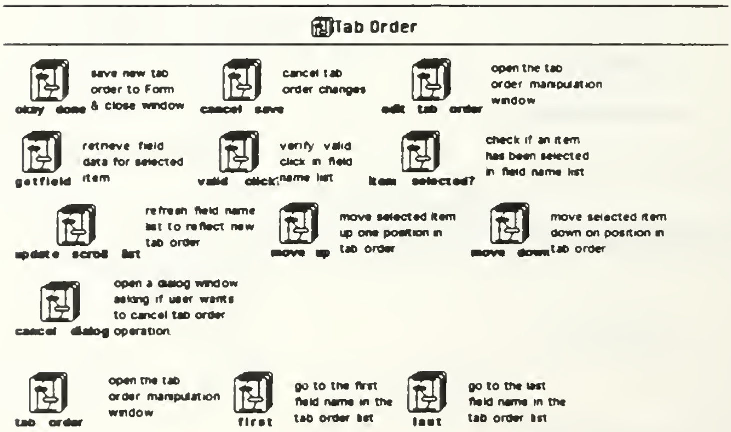
"/Close"
close method

• •
idle method

• •
key method

(<<Scroll L...
item list

(<<Field>> ...
temp fieldinfo



Tab Order

okay done
save new tab order to Form & close window

cancel save
cancel tab order changes

edit tab order
open the tab order manipulation window

getfield
retrieve field data for selected item

valid click?
verify valid click in field name list

item selected?
check if an item has been selected in field name list

update scroll list
refresh field name list to reflect new tab order

move up
move selected item up one position in tab order

move down
move selected item down on position in tab order

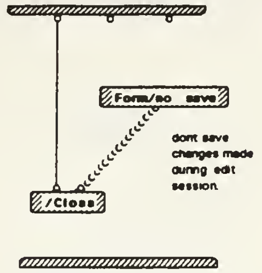
cancel dialog
open a dialog window asking if user wants to cancel tab order operation.

tab order
open the tab order manipulation window

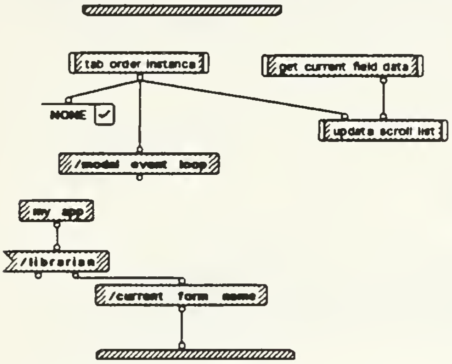
first
go to the first field name in the tab order list

last
go to the last field name in the tab order list

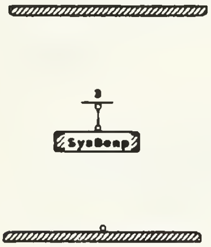
▨Tab Order/cancel save 1:1



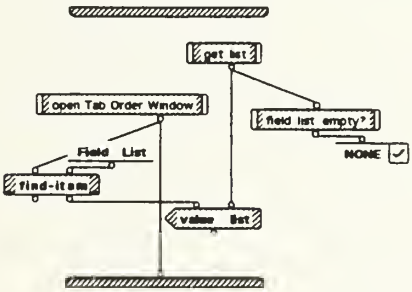
▨Tab Order/edit tab order 1:2



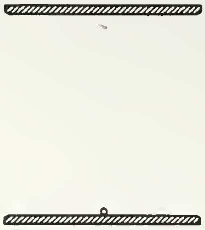
▨Tab Order/edit tab order 2:2



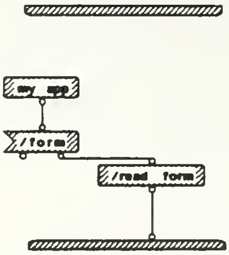
▨Tab Order/edit tab order 1:2tab order instance 1:2



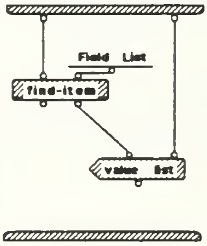
▨Tab Order/edit tab order 1:2tab order instance 2:2



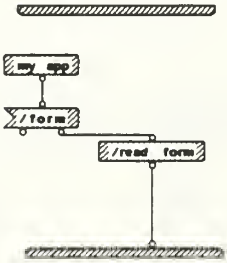
▨Tab Order/edit tab order 1:2get current field data 1:1



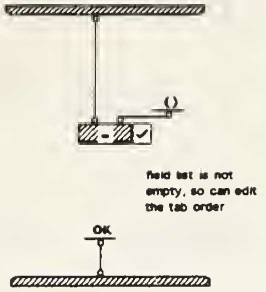
▨Tab Order/edit tab order 1:2update scroll list 1:1



▨Tab Order/edit tab order 1:2tab order instance 1:2get list 1:1

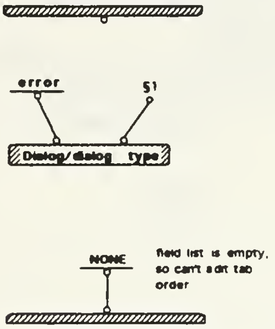


▨Tab Order/edit tab order t:2tab order instance t:2field list empty? t:2



field list is not empty, so can edit the tab order

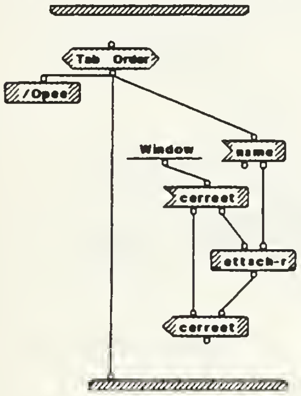
▨Tab Order/edit tab order t:2tab order instance t:2field list empty? 2:2



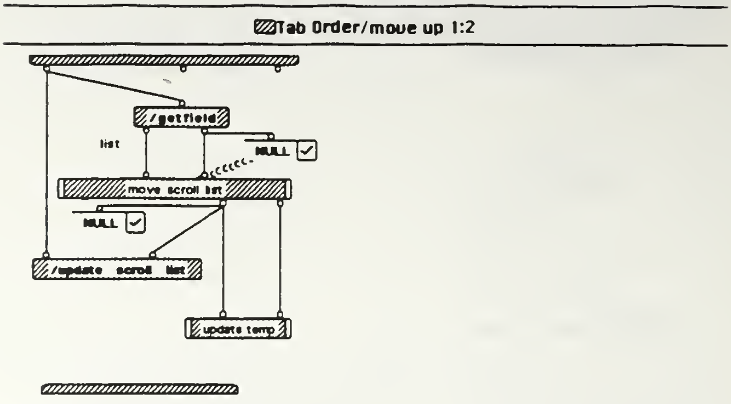
field list is empty, so can't edit tab order

§1 There are no fields defined for the current form

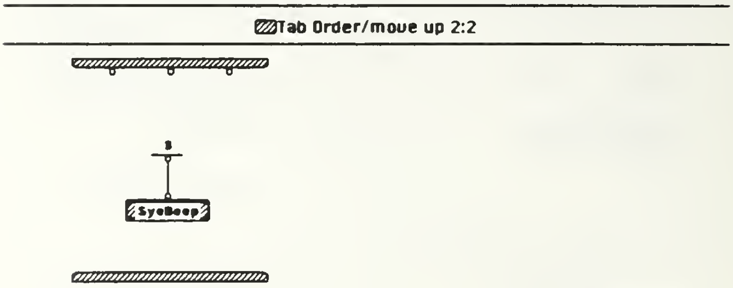
▨Tab Order/edit tab order 1:2tab order instance t:2open Tab Order Window t:1



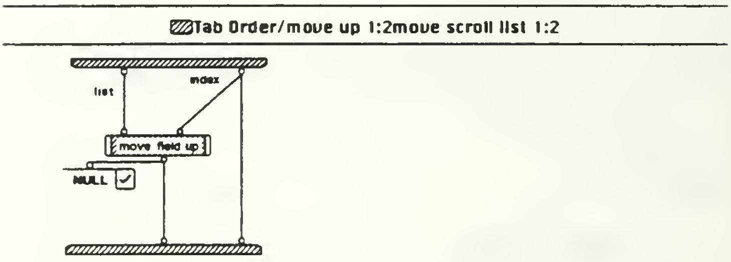
set the attribute "current" in class Window to reflect the instance of "Tab Order" This attribute acts like a stack, maintaining a list of open windows, with the left-most name in the list being at the top of the stack (ie - the topmost window)



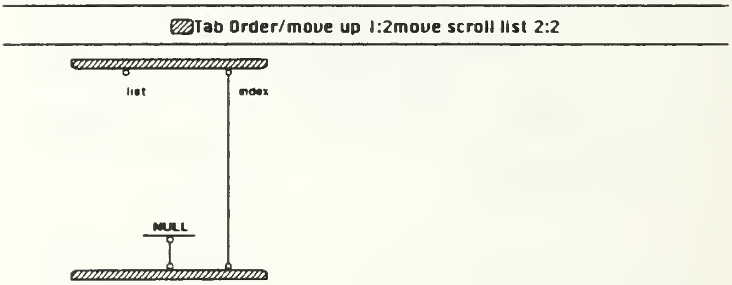
▨Tab Order/move up 1:2



▨Tab Order/move up 2:2

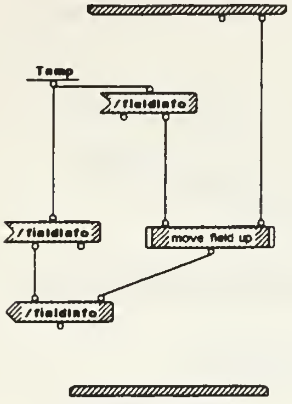


▨Tab Order/move up 1:2move scroll list 1:2

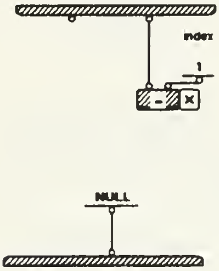


▨Tab Order/move up 1:2move scroll list 2:2

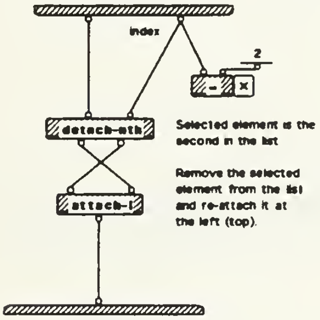
▨Tab Order/move up 1:2update temp 1:1



▨Tab Order/move up 1:2move scroll list 1:2move field up 1:3



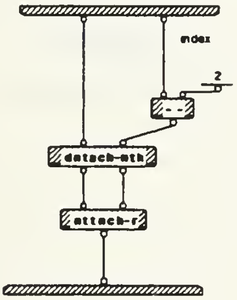
▨Tab Order/move up 1:2move scroll list 1:2move field up 2:3



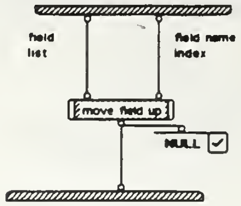
Selected element is the second in the list

Remove the selected element from the list and re-attach it at the left (top).

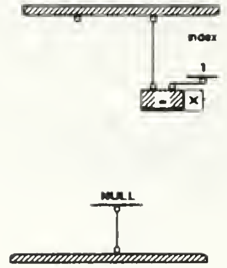
▨Tab Order/move up 1:2move scroll list 1:2move field up 3:3



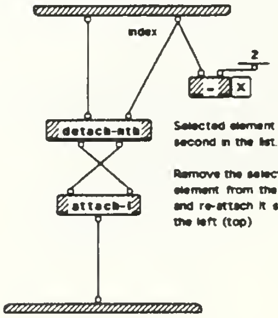
▨Tab Order/move up 1:2update temp 1:1move field up 1:1



▨Tab Order/move up 1:2updale lemp 1:1move field up 1:1move field up 1:3



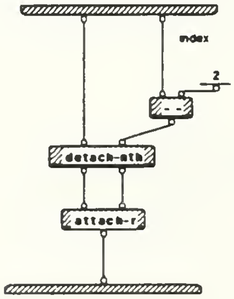
▨Tab Order/move up 1:2updale lemp 1:1move field up 1:1move field up 2:3



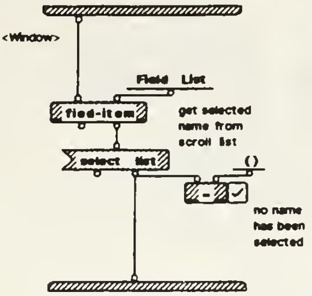
Selected element is the second in the list.

Remove the selected element from the list and re-attach it at the left (top)

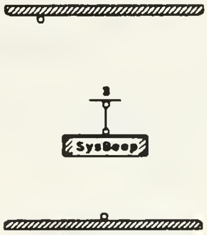
▨Tab Order/move up 1:2updale lemp 1:1move field up 1:1move field up 3:3



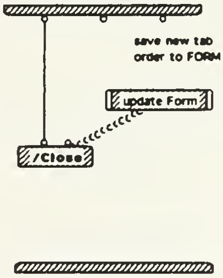
▨Tab Order/Item selected? 1:2



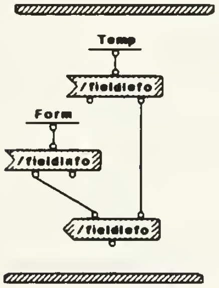
▨Tab Order/Item selected? 2:2



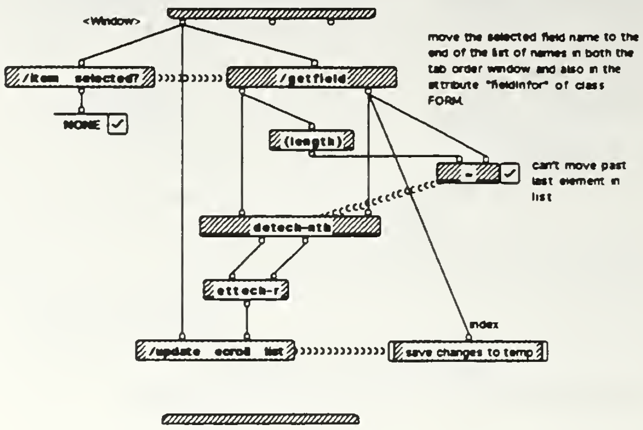
▨Tab Order/okay done 1:1



▨Tab Order/okay done 1:1update Form 1:1



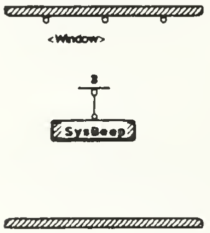
▨Tab Order/last 1:2



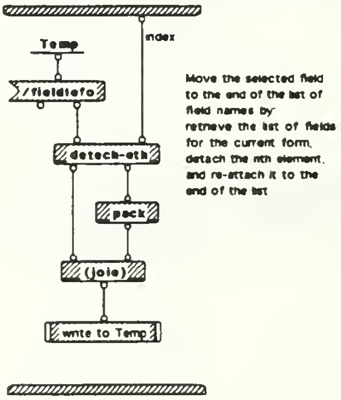
move the selected field name to the end of the list of names in both the tab order window and also in the attribute "fieldinfor" of class FORM.

can't move past last element in list

▨Tab Order/last 2:2

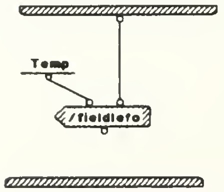


▨Tab Order/last 1:2save changes to temp 1:1

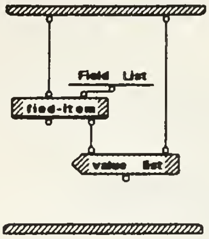


Move the selected field to the end of the list of field names by: retrieve the list of fields for the current form, detach the nth element, and re-attach it to the end of the list

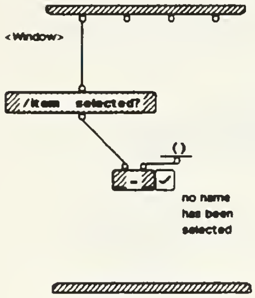
▨Tab Order/last 1:2save changes to temp 1:1write to Temp 1:1



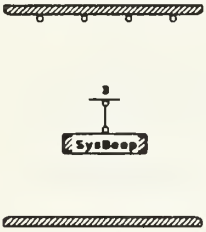
▨Tab Order/update scroll list 1:1



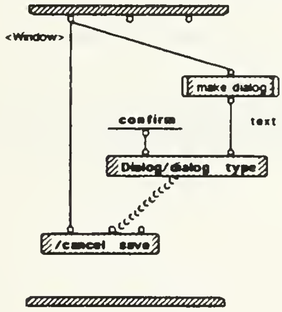
▨Tab Order/valid click? 1:2



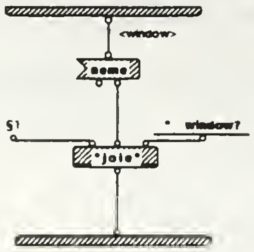
▨Tab Order/valid click? 2:2



▨Tab Order/cancel dialog 1:1

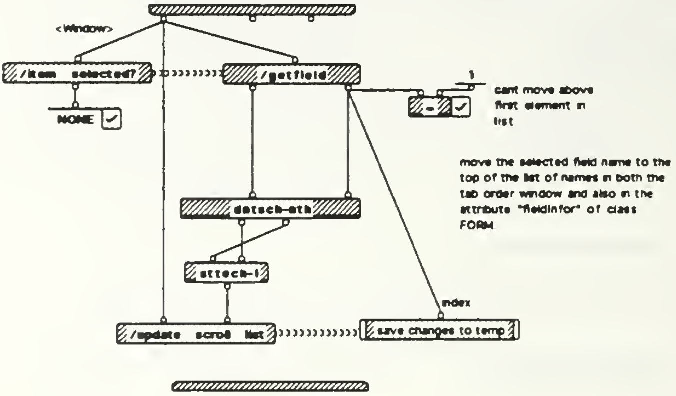


▨Tab Order/cancel dialog 1:1make dialog 1:1



§1 Unsaved changes will be lost. Continue with selection and close "

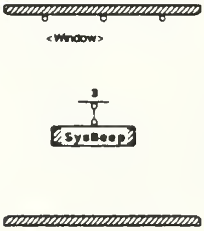
▨Tab Order/first 1:2



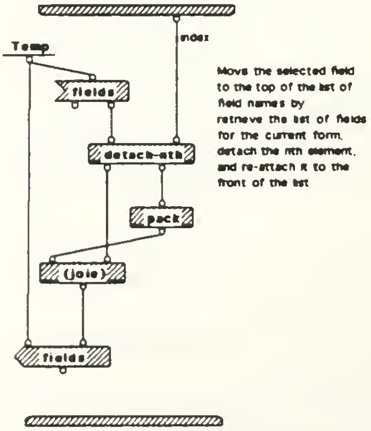
cant move above first element in list

move the selected field name to the top of the list of names in both the tab order window and also in the attribute "fieldinfor" of class FORM.

▨Tab Order/first 2:2

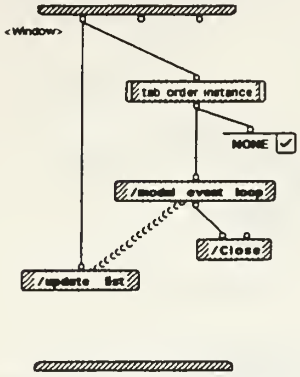


▨Tab Order/first 1:2save changes to temp 1:1

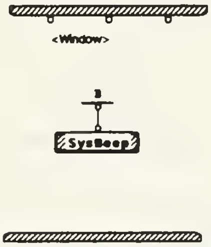


Move the selected field to the top of the list of field names by retrieve the list of fields for the current form, detach the nth element, and re-attach it to the front of the list

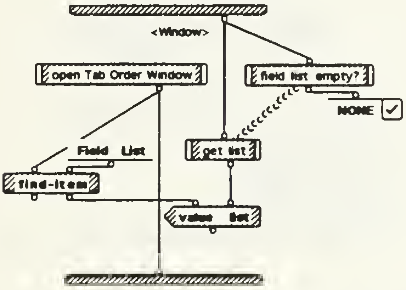
▨Tab Order/tab order 1:2



▨Tab Order/tab order 2:2

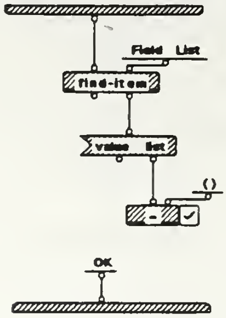


▨Tab Order/tab order 1:2tab order instance 1:2

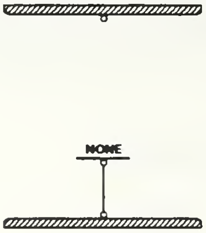


▨Tab Order/tab order 1:2tab order instance 2:2

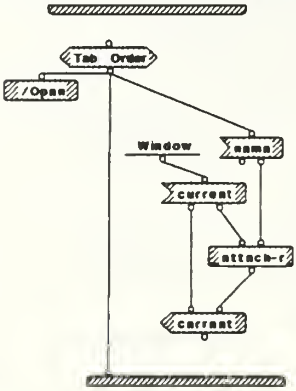
▨Tab Order/tab order 1:2tab order instance 1:2field list empty? 1:2



▨Tab Order/tab order 1:2tab order instance 1:2field list empty? 2:2

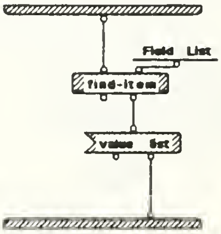


▨Tab Order/tab order 1:2tab order instance 1:2open Tab Order Window 1:1

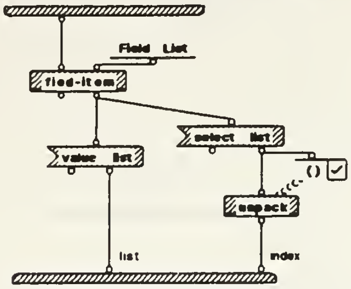


set the attribute "current" in class Window to reflect the instance of "Tab Order" This attribute acts like a stack, maintaining a list of open windows, with the left-most name in the list being at the top of the stack (ie - the topmost window)

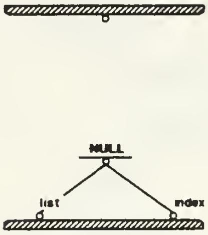
▨Tab Order/tab order 1:2tab order instance 1:2get list 1:1



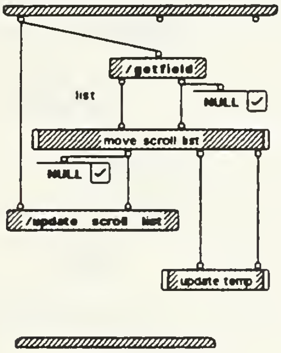
▨Tab Order/getfield 1:2



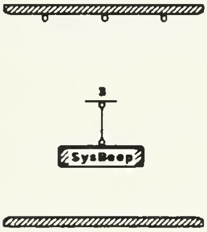
▨Tab Order/getfield 2:2



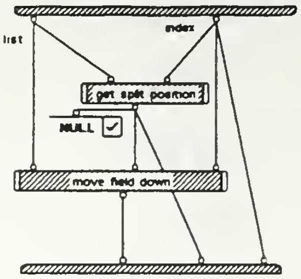
▨Tab Order/move down 1:2



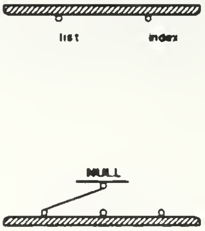
▨Tab Order/move down 2:2



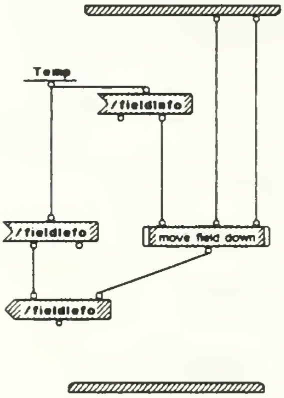
▨Tab Order/move down 1:2move scroll list 1:2



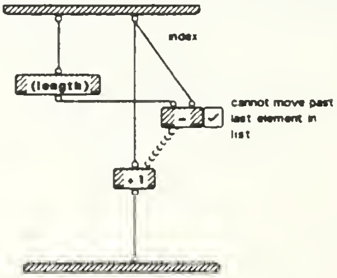
▨Tab Order/move down 1:2move scroll list 2:2



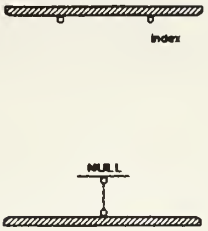
▨Tab Order/move down 1:2update temp 1:1



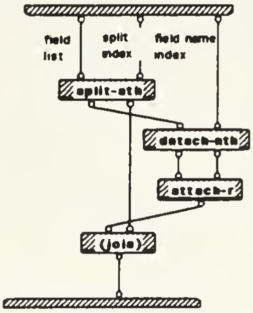
▨Tab Order/move down 1:2move scroll list 1:2get split position 1:2



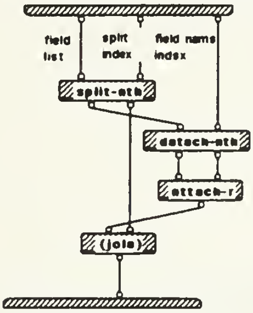
▨Tab Order/move down 1:2move scroll list 1:2get split position 2:2

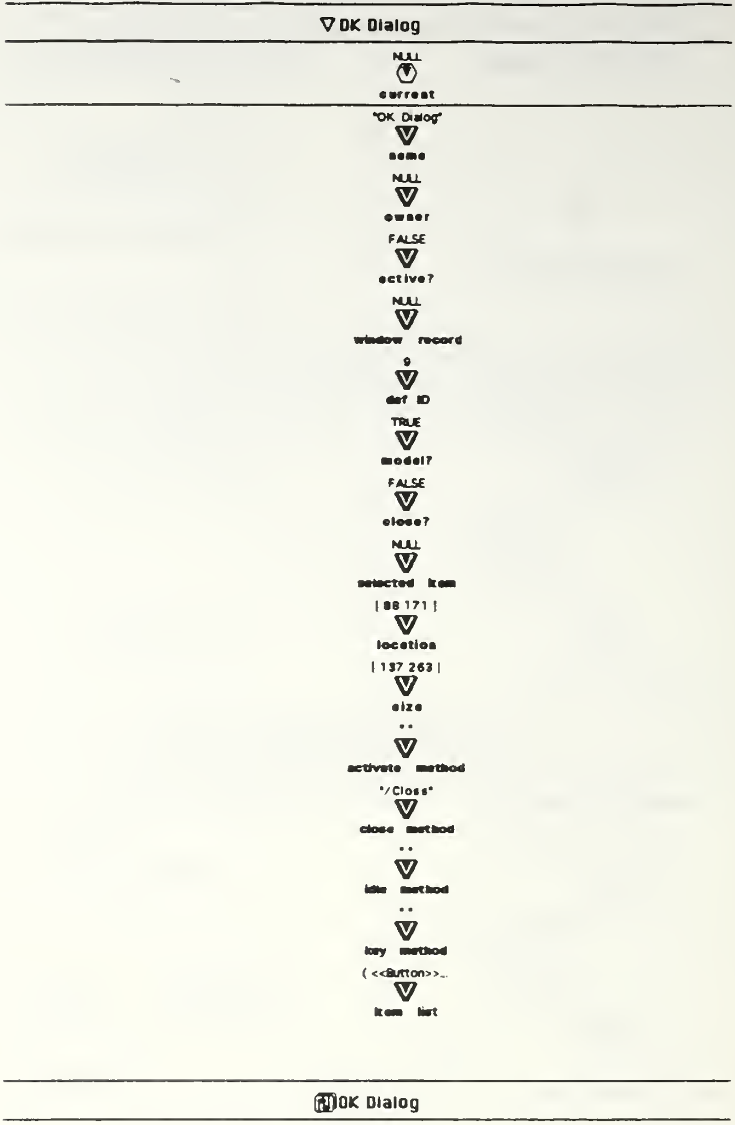


▨Tab Order/move down 1:2move scroll list 1:2move field down 1:1



▨Tab Order/move down 1:2update temp 1:1move field down 1:1





∇ OK Dialog

NULL
current

"OK Dialog"
name

NULL
owner

FALSE
active?

NULL
window record

9
def ID

TRUE
modal?

FALSE
close?

NULL
selected item

[88 171]
location

[137 263]
size

• •
activate method

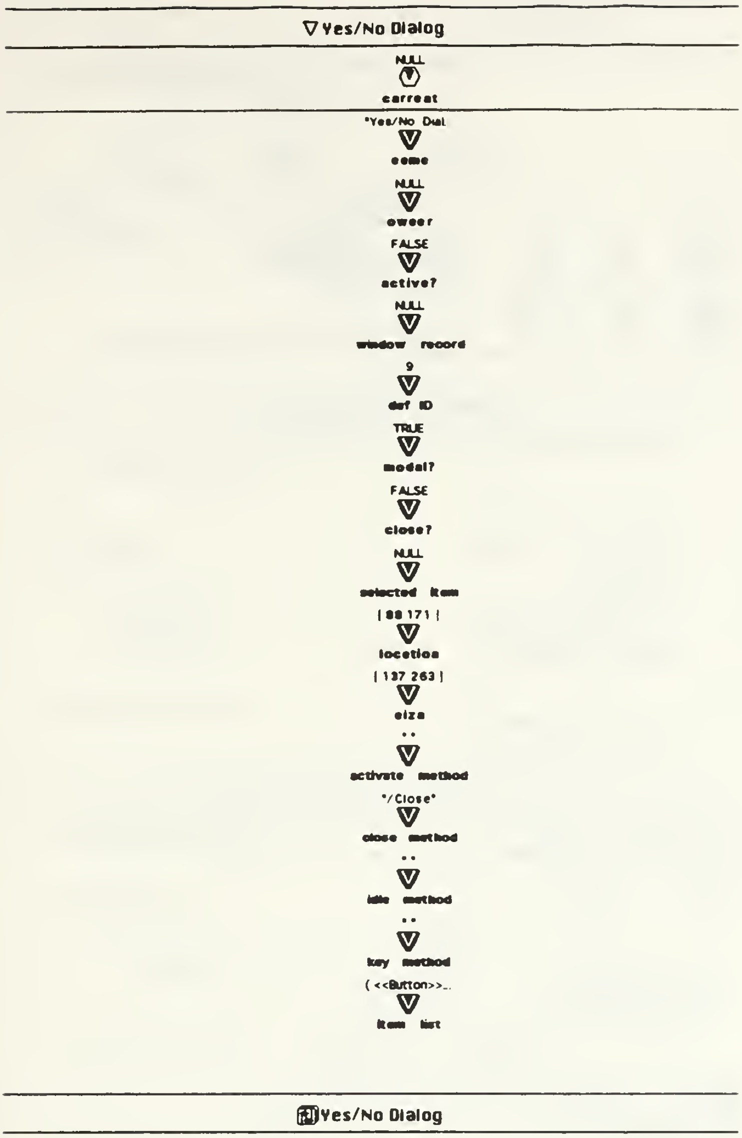
"/Close"
close method

• •
idle method

• •
key method

(<<Button>>...
item list

OK Dialog



∇ Yes/No Dialog

NULL
current

"Yes/No Dial
name

NULL
owner

FALSE
active?

NULL
window record

9
def ID

TRUE
modal?

FALSE
close?

NULL
selected item

(88 171)
location

(137 263)
size

• •
activate method

"/Close"
close method

• •
idle method

• •
key method

(<<Button>>_
item list

Yes/No Dialog

∇Temp

()
fields

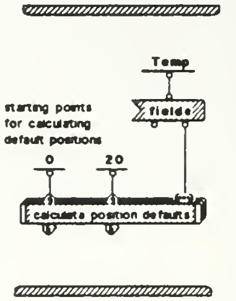
list of field names, types, attribute & position o each field on the IE Canvas
((<name type attribute posit>))

Temp

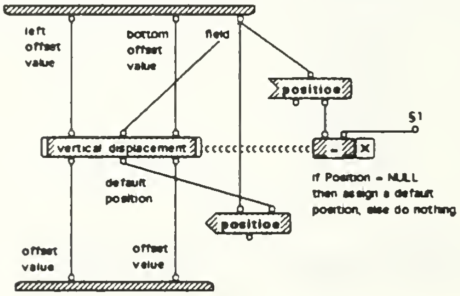
add field · delete field · find field · replace field · field names

field list · reset temp · revert temp · default position

▨Temp/default position 1:1

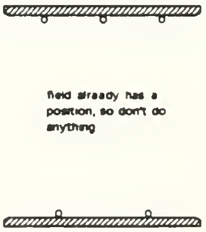


▨Temp/default position 1:1calculate position defaults 1:2

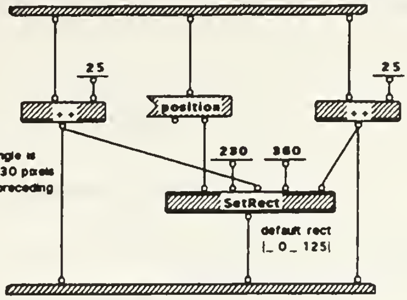


$1 [-1 -1 -1 -1]

▨Temp/default position 1:1calculate position defaults 2:2

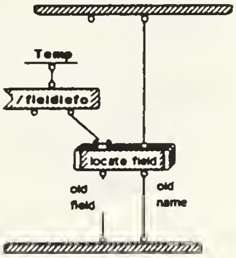


Temp/default position 1:1 calculate position defaults 1:2 vertical displacement 1 1

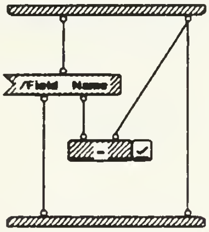


each rectangle is positioned 30 pixels below the preceding one.

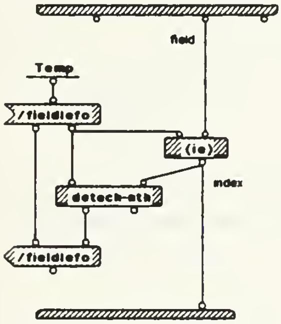
Temp/find field 1:1



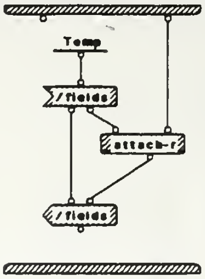
Temp/find field 1:locate field 1:1



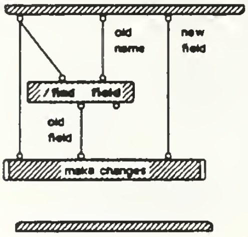
Temp/delete field 1:1



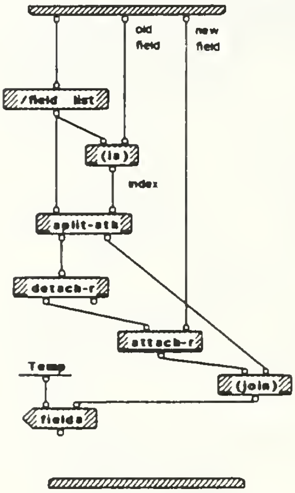
▨Temp/add field 1:1



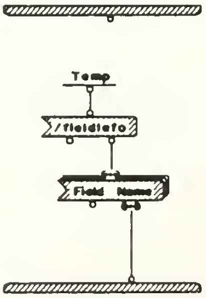
▨Temp/replace field 1:1



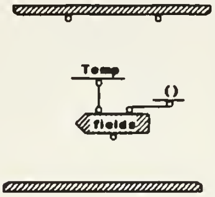
▨Temp/replace field 1:1make changes 1:1



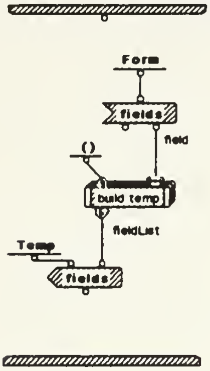
▨Temp/field names 1:1



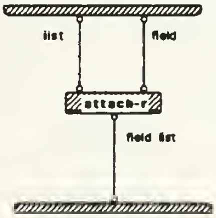
▨Temp/reset temp 1:1



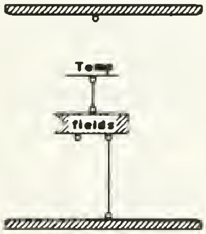
▨Temp/revert temp 1:1

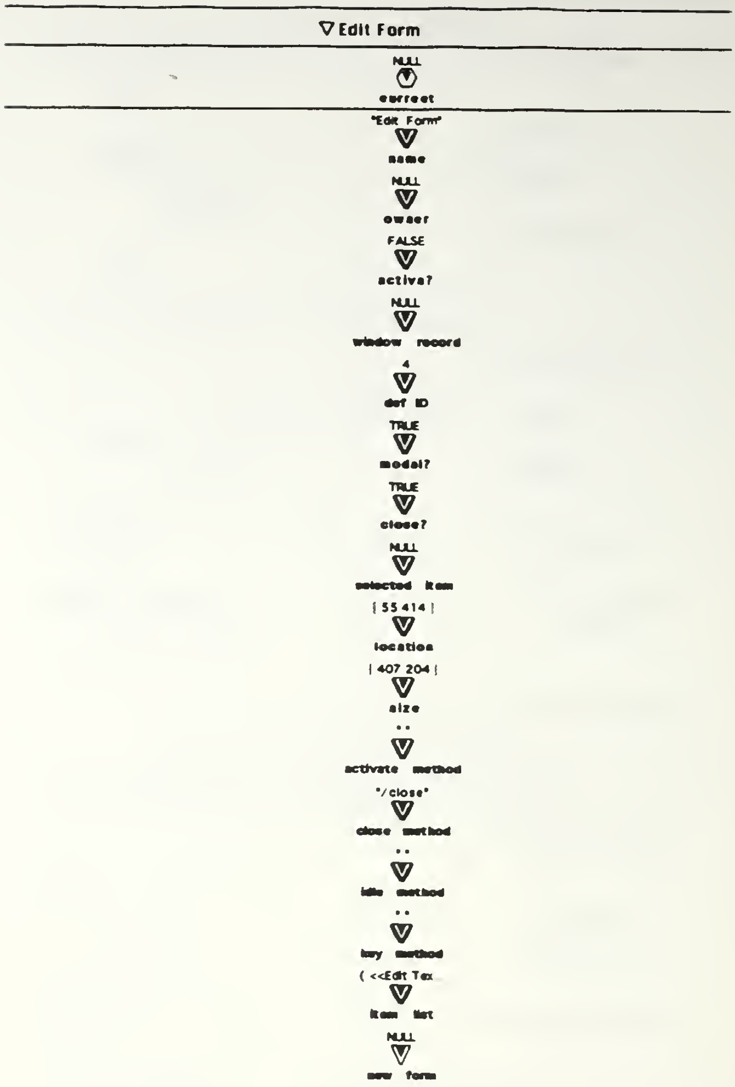


▨Temp/revert temp 1:1build temp 1:1



▨Temp/field list 1:1





∇ Edit Form

NULL
current

"Edit Form"
name

NULL
owner

FALSE
active?

NULL
window record

4
def ID

TRUE
modal?

TRUE
close?

NULL
selected item

(55 414)
location

(407 204)
size

• •
activate method

"/close"
close method

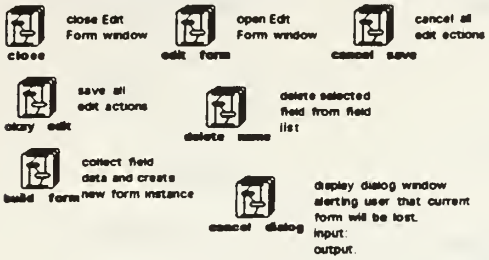
• •
idle method

• •
key method

(<<Edit Tex
item list

NULL
new form

Edit Form



close — close Edit Form window

edit form — open Edit Form window

cancel save — cancel all edit actions

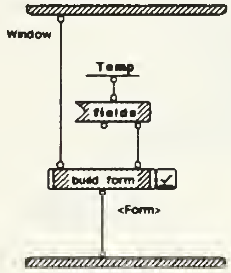
okay edit — save all edit actions

delete name — delete selected field from field list

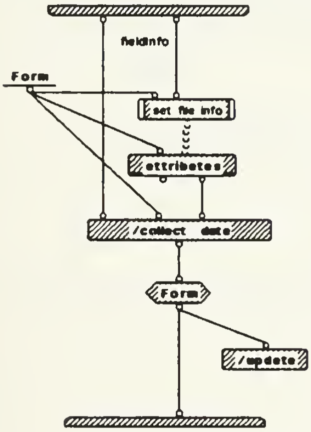
build form — collect field data and create new form instance

cancel dialog — display dialog window alerting user that current form will be lost.
input:
output:

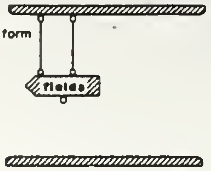
Edit Form/build form 1:1



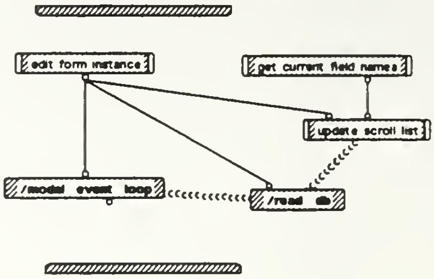
Edit Form/build form 1:1build form 1:1



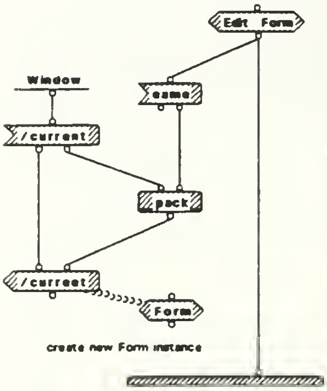
▨Edit Form/build form 1:1build form 1:1set file info 1:1



▨Edit Form/edit form 1:1



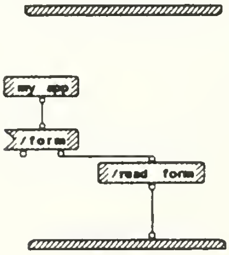
▨Edit Form/edit form 1:1edit form instance 1:1



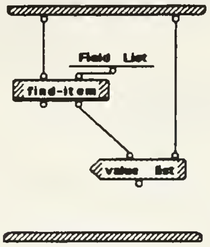
create new Form instance

set the attribute "current" in class Window to reflect the instance of "New Form" This attribute acts like a stack, maintaining a list of open windows, with the left-most name in the list being at the top of the stack (ie - the topmost window)

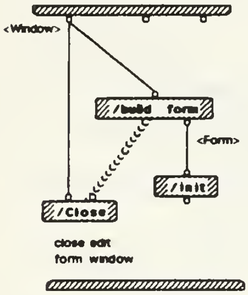
▨Edit Form/edit form 1:1get current field names 1:1



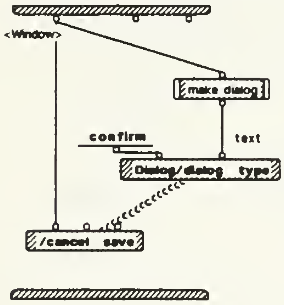
▨Edit Form/edit form 1:update scroll list 1:1



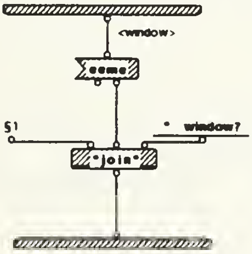
▨Edit Form/okay edit 1:1



▨Edit Form/cancel dialog 1:1

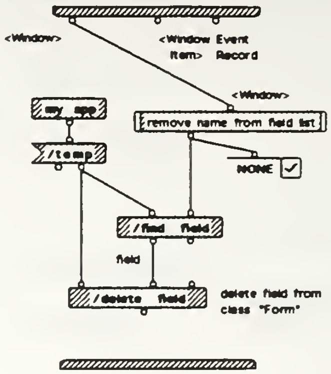


▨Edit Form/cancel dialog 1:make dialog 1:1

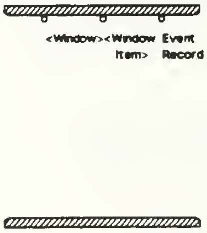


§1 Unsaved changes will be lost Continue with selection and close "

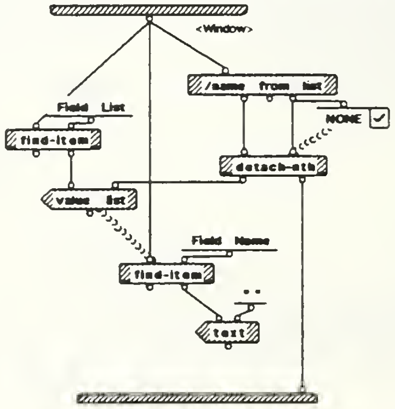
▨Edit Form/delete name 1:2



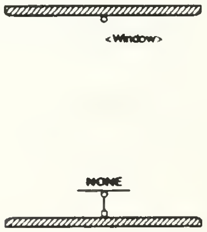
▨Edit Form/delete name 2:2



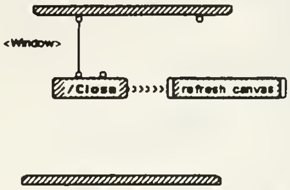
▨Edit Form/delete name 1:2remove name from field list 1:2



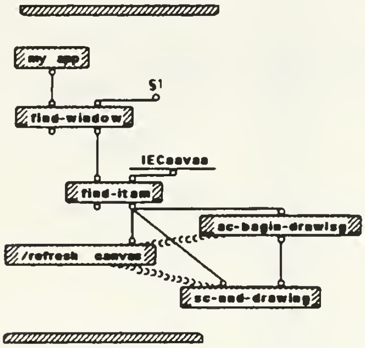
▨Edit Form/delete name 1:2remove name from field list 2:2



▨Edit Form/close 1:1

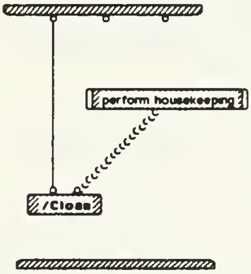


▨Edit Form/close 1:1refresh canvas 1:1

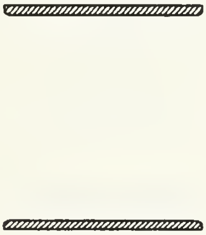


$1 Interface Editor Window

▨Edit Form/cancel save 1:1



▨Edit Form/cancel save 1:1perform housekeeping 1:1



∇ Preferences

(( ) "Desig.
current

"User Prefer
name

NULL
owner

FALSE
active?

NULL
window record

4
def ID

TRUE
modal?

TRUE
close?

NULL
selected item

| 142 420 |
location

| 316 199 |
size

• •
activate method

"/Close"
close method

• •
idle method

• •
key method

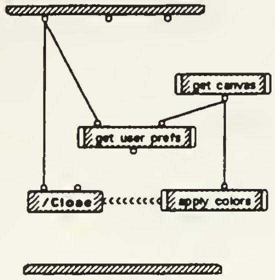
(<<Button>>_
item list

Preferences

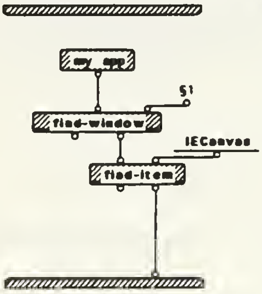
user prefs — open Prefs window and get user prefs

okay prefs — apply user prefs to canvas

▨Preferences/okay prefs 1:1

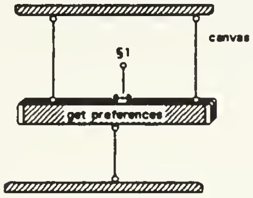


▨Preferences/okay prefs 1:1get canvas 1:1



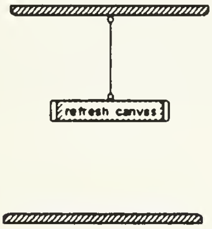
§1 Interface Editor Window

▨Preferences/okay prefs 1:1get user prefs 1:1

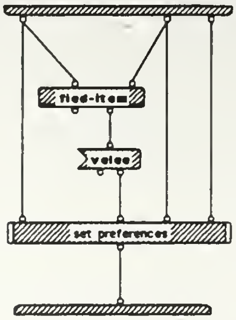


§1 (background foreground)

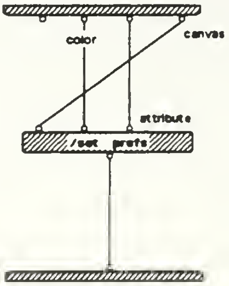
▨Preferences/okay prefs 1:1apply colors 1:1



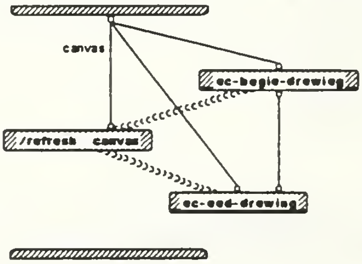
▨Preferences/okay prefs 1:1get user prefs 1:1get preferences 1:1



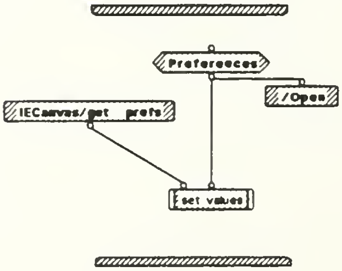
▨Preferences/okay prefs 1:1get user prefs 1:1get preferences 1:1set preferences 1:1



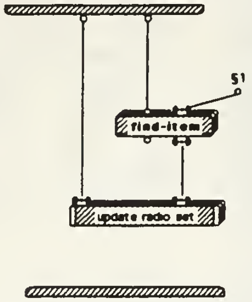
▨Preferences/okay prefs 1:1apply colors 1:1refresh canvas 1:1



▨Preferences/user prefs 1:1

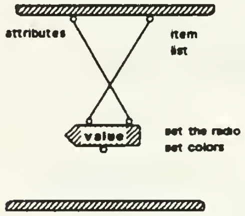


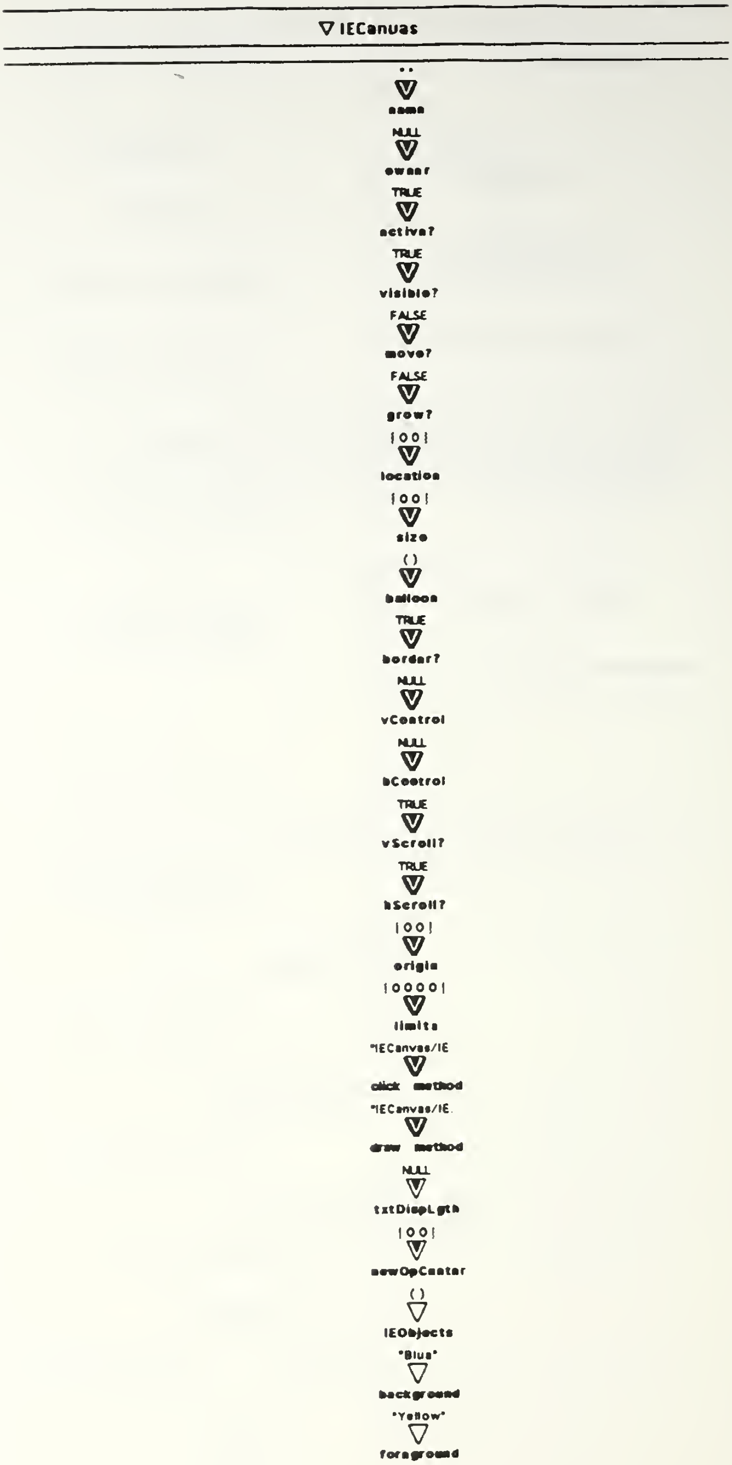
▨Preferences/user prefs 1:1set values 1:1



§1. (background foreground)

▨Preferences/user prefs 1:1set values 1:1update radio set 1:1

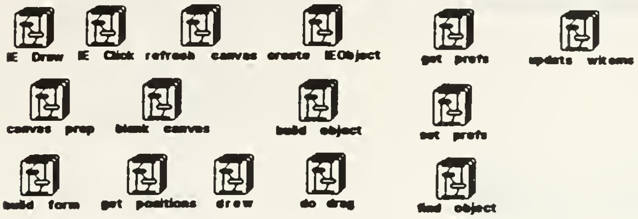




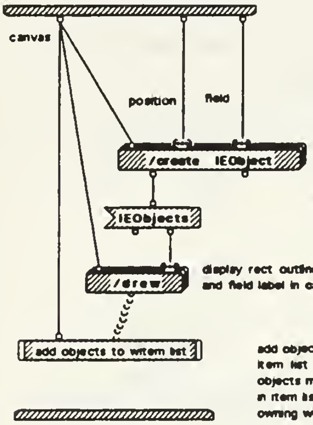
∇ IECanvas

```
    • •
     ∇
   name
   NULL
     ∇
   owner
   TRUE
     ∇
  active?
   TRUE
     ∇
  visible?
   FALSE
     ∇
   move?
   FALSE
     ∇
   grow?
  | 0 0 |
     ∇
 location
  | 0 0 |
     ∇
   size
    ( )
     ∇
  balloon
   TRUE
     ∇
  border?
   NULL
     ∇
 vControl
   NULL
     ∇
 hControl
   TRUE
     ∇
 vScroll?
   TRUE
     ∇
 hScroll?
  | 0 0 |
     ∇
  origin
| 0 0 0 0 |
     ∇
  limits
"IECanvas/IE
     ∇
click method
"IECanvas/IE.
     ∇
draw method
   NULL
     ∇
txtDispLgth
  | 0 0 |
     ∇
newOpCenter
    ( )
     ∇
 IEObjects
  "Blue"
     ∇
background
 "Yellow"
     ∇
foreground
```

IECanvas



IECanvas/build object 1:2



generate instances
for each field type
and add to canvas
object list

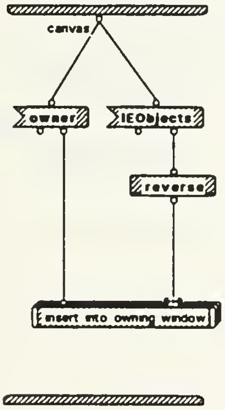
display rect outline
and field label in canvas

add objects to window
item list
objects must be
in item list of
owning window to
be visible

IECanvas/build object 2:2

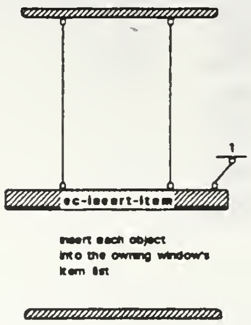
no fields have been defined
for this form. no action
required

IECanvas/build object 1:2add objects to witem list 1:1

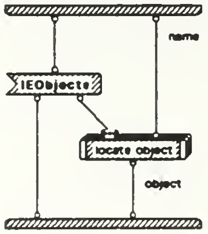


reverse list
to preserve
tab order

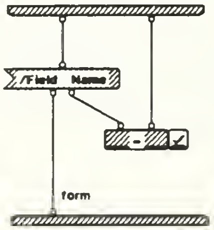
IECanvas/build object 1:2add objects to witem list 1:1insert into owning window 1:1



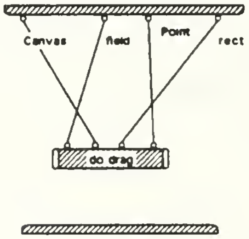
IECanvas/find object 1:1



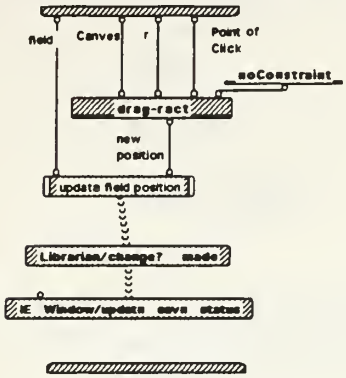
IECanvas/find object 1:1locate object 1:1



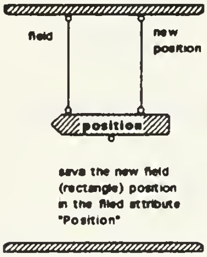
IECanvas/do drag 1:1



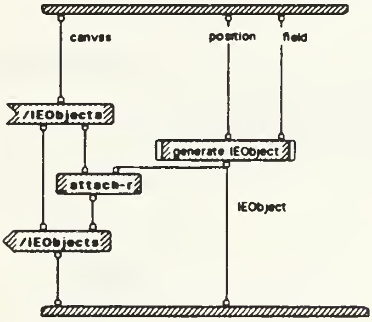
▨IECanvas/do drag 1:1do drag 1:1



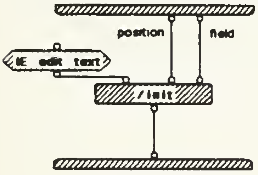
▨IECanvas/do drag 1:1do drag 1:1update field position 1:1


field
new position
position
save the new field (rectangle) position in the filed attribute "Position"

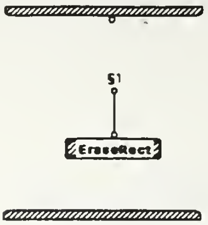
▨IECanvas/create IEObject 1:1



▨IECanvas/create IEObject 1:1generate IEObject 1:1

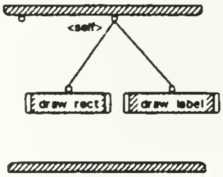


▨IECanvas/canvas prep 1:1

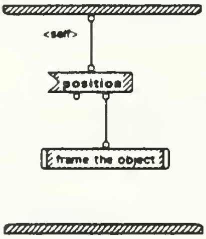


$1. {0 0 32000 32000}

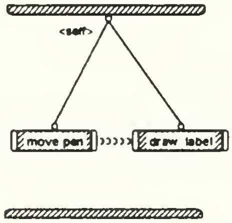
▨IECanvas/draw 1:1



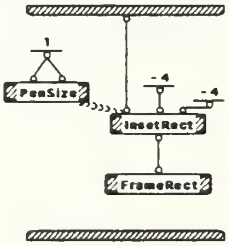
▨IECanvas/draw 1:1draw rect 1:1



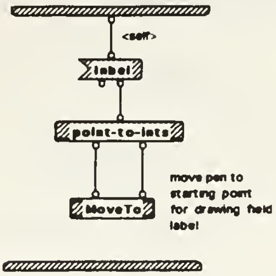
▨IECanvas/draw 1:1draw label 1:1



▨IECanvas/draw 1:1draw rect 1:1frame the object 1:1

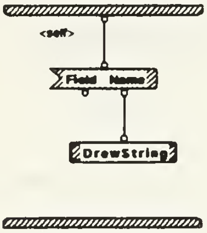


▨IECanvas/draw 1:1draw label 1:1move pen 1:1

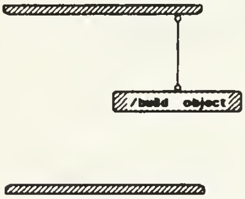


move pen to starting point for drawing field label

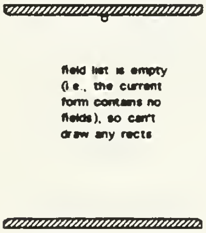
▨IECanvas/draw 1:1draw label 1:1draw label 1:1



▨IECanvas/build form 1:2

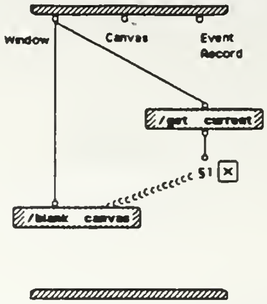


▨IECanvas/build form 2:2



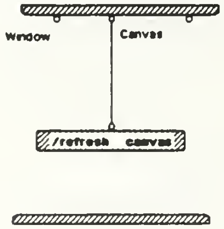
field list is empty (i.e., the current form contains no fields), so can't draw any rects

▨IECanvas/IE Draw 1:2

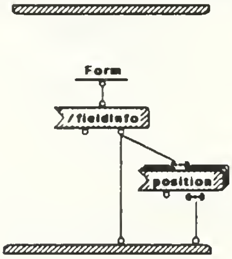


§1 <no currently active form>

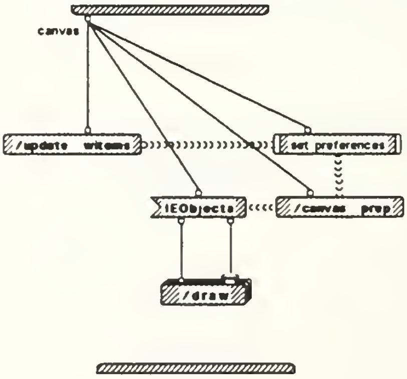
▨IECanvas/IE Draw 2:2



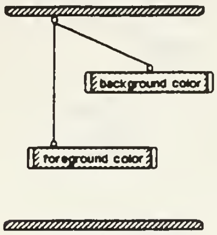
▨IECanvas/get positions 1:1



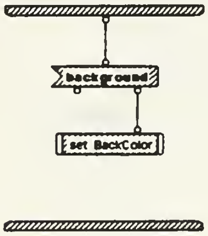
▨IECanvas/refresh canvas 1:1



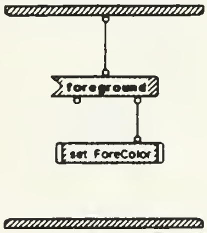
▨IECanvas/refresh canvas 1:1set preferences 1:1



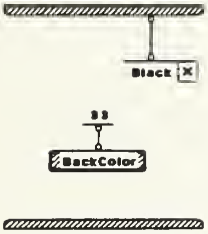
▨IECanvas/refresh canvas 1:1set preferences 1:1background color 1:1



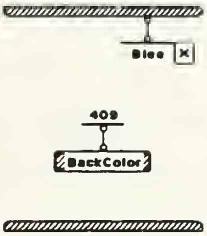
▨IECanvas/refresh canvas 1:1set preferences 1:1foreground color 1:1



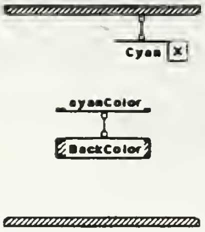
▨IECanvas/refresh canvas 1:1set preferences 1:1background color 1:1set BackColor 1:8



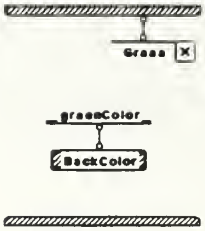
▨IECanvas/refresh canvas 1:1set preferences 1:1background color 1:1set BackColor 2:8



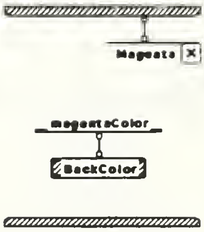
▨ IECanvas/refresh canvas 1:1 set preferences 1:1 background color 1:1 set BackColor 3:8



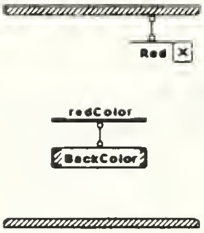
▨ IECanvas/refresh canvas 1:1 set preferences 1:1 background color 1:1 set BackColor 4:8



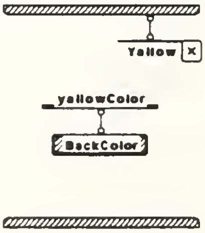
▨ IECanvas/refresh canvas 1:1 set preferences 1:1 background color 1:1 set BackColor 5:8



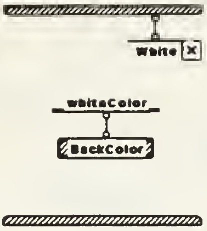
▨ IECanvas/refresh canvas 1:1 set preferences 1:1 background color 1:1 set BackColor 6:8



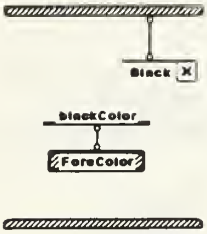
▨ IECanvas/refresh canvas 1:1 set preferences 1:1 background color 1:1 set BackColor 7:8



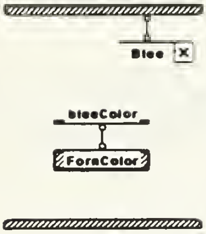
IECanvas/refresh canvas 1:1 set preferences 1:1 background color 1:1 set BackColor 8:8



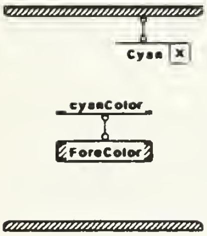
IECanvas/refresh canvas 1:1 set preferences 1:1 foreground color 1:1 set ForeColor 1:8



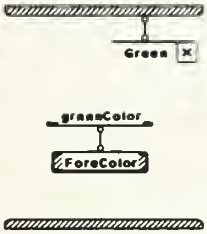
IECanvas/refresh canvas 1:1 set preferences 1:1 foreground color 1:1 set ForeColor 2:8



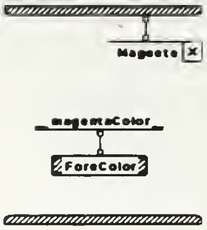
IECanvas/refresh canvas 1:1 set preferences 1:1 foreground color 1:1 set ForeColor 3:8



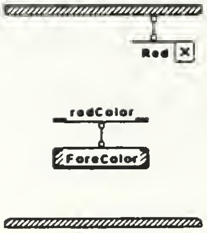
IECanvas/refresh canvas 1:1 set preferences 1:1 foreground color 1:1 set ForeColor 4:8



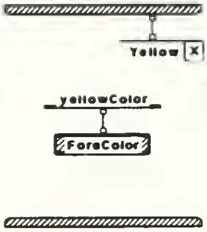
▨IECanvas/refresh canvss 1:1set preferences 1:1 foreground color 1:1set ForeColor 5:8



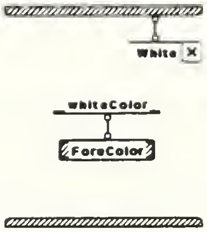
▨IECanvas/refresh canvss 1 1set preferences 1:1 foreground color 1.1set ForeColor 6:8



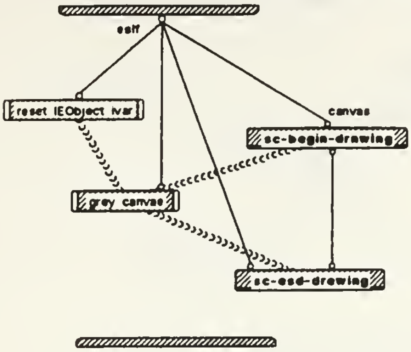
▨IECanvas/rsfresh canvas 1:1set preferences 1 1foreground color 1 1set ForeColor 7.8



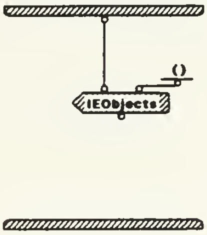
▨IECanvss/refresh canvas 1:1set preferences 1:1foreground color 1:1set ForeColor 8 8



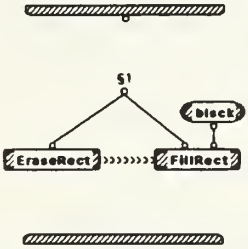
▨IECanvas/blank canvas 1:1



▨IECanvas/blank canvas 1:1reset IEObject ivar 1:1

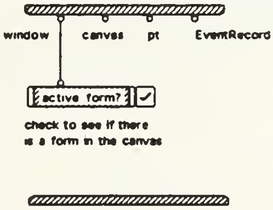


▨IECanvas/blank canvas 1:1grey canvas 1:1

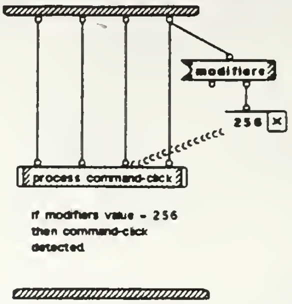


$1 (0 0 32000 32000)

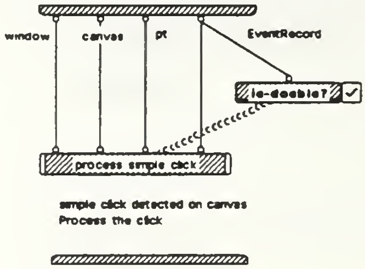
▨IECanvas/IE Click 1:4



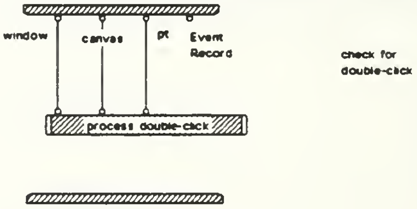
IECanvas/IE Click 2:4



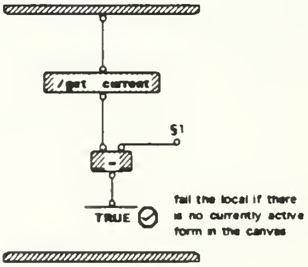
IECanvas/IE Click 3:4



IECanvas/IE Click 4:4

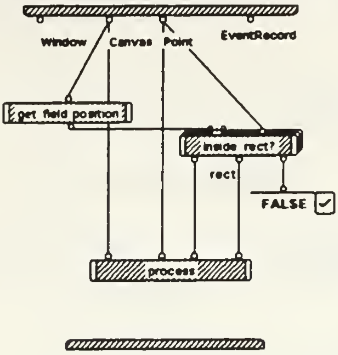


IECanvas/IE Click 1:4active form? 1:1

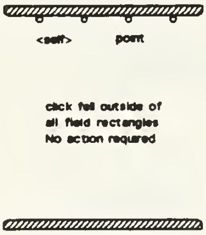


§1 <no currently active form>

▨IECanvas/IE Click 2:4process command-click 1:2

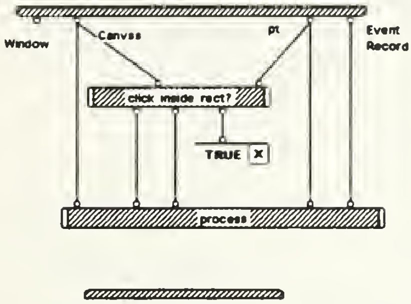


▨IECanvas/IE Click 2:4process command-click 2:2

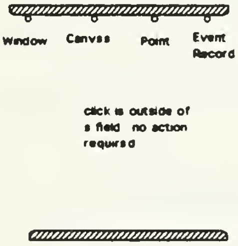


click fell outside of
all field rectangles
No action required

▨IECanvas/IE Click 3:4process simple click 1:2

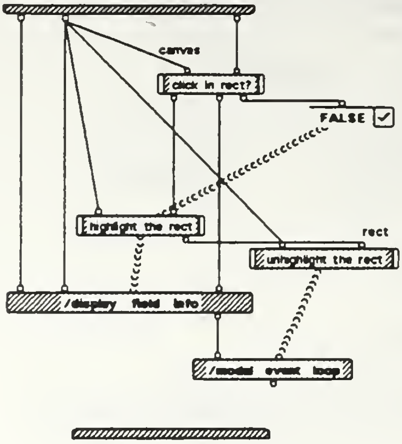


▨IECanvas/IE Click 3:4process simple click 2:2

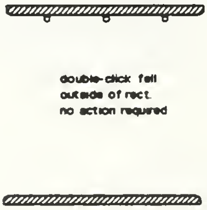


click is outside of
a field no action
required

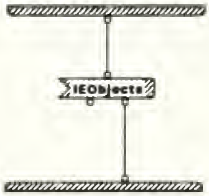
▨IECanvas/IE Click 4:4process double-click 1:2



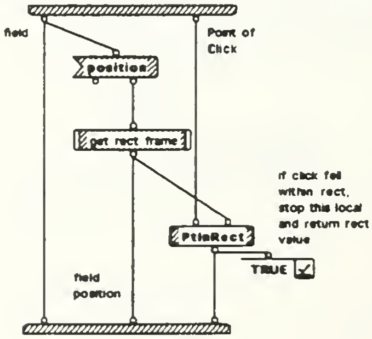
▨IECanvas/IE Click 4:4process double-click 2:2



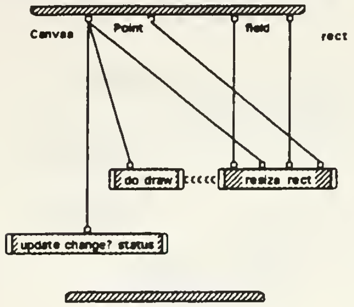
▨IECanvas/IE Click 2:4process command-click 1:2get field position 1:1



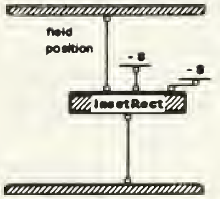
▨IECanvas/IE Click 2:4process command-click 1:2inside rect? 1:1



▨IECanvas/IE Click 2:4process command-click 1:2process 1:1

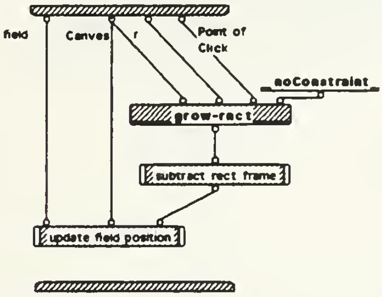


▨IECanvas/IE Click 2:4process command-click 1:2inside rect? 1 1get rect frame 1 1

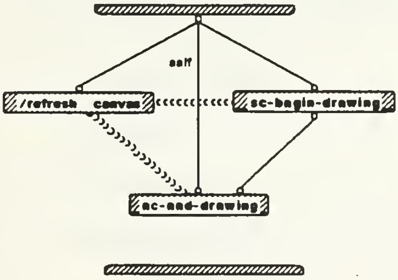


on canvas, rect is framed
add the InsetRect amount
to Position to ensure proper
rect is presented to PtInRect

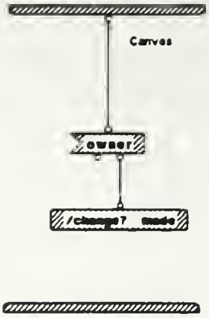
▨IECanvas/IE Click 2:4process command-click 1:2process 1:1resize rect 1:1



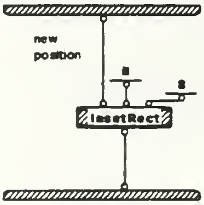
▨IECanvas/IE Click 2:4process command-click 1:2process 1:1do draw 1:1



▨IECanvas/IE Click 2 4process command-click 1:2process 1'1 update change? status 1'1

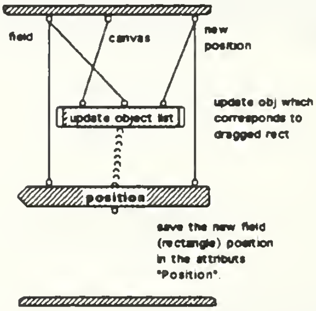


▨IECanvas/IE Click 2:4process command-click 1:2process 1:1resize rect 1 1subtract rect frame 1.1



rect was enlarged when
originally drawn to provide
separation between edit text
and rect frame must remove
this extrs area when re-drawing
rect, or else rect will grow

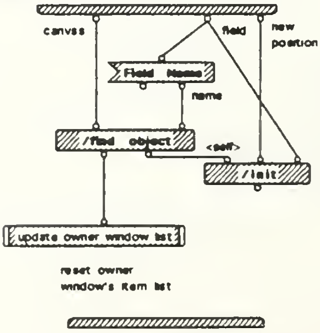
▨IECanvas/IE Click 2:4process command-click 1 2process 1:1resize rect 1:1update field position 1 1



update obj which
corresponds to
dragged rect

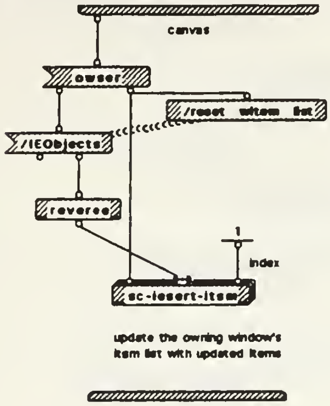
save the new field
(rectangle) position
in the attribute
"Position".

▨IECanvas/IE Click 2 4process command-click 1:2process 1 1resize rect 1 1update field position 1 1update object
▨st 1 1

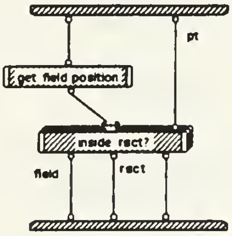


reset owner
window's item list

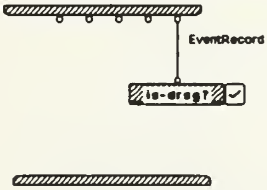
▨IECanvas/IE Click 2:4process command-click 1:2process 1:1resize rect 1:1update field position 1:1update object
▨st 1:1update owner window list 1:1



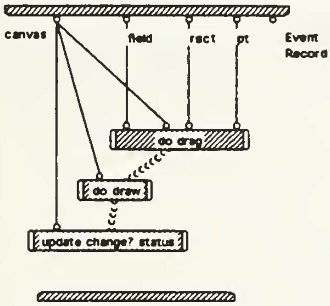
▨IECanvas/IE Click 3:4process simple click 1:2click inside rect? 1:1



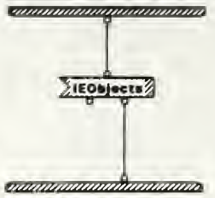
▨IECanvas/IE Click 3:4process simple click 1:2process 1:2



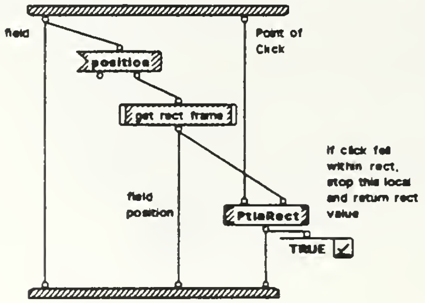
▨IECanvas/IE Click 3:4process simple click 1:2process 2:2



▨IECanvas/IE Click 3:4process simple click 1:2click inside rect? 1:1get field position 1:1

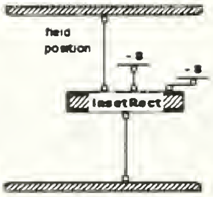


▨IECanvas/IE Click 3:4process simple click 1:2click inside rect? 1:1inside rect? 1:1



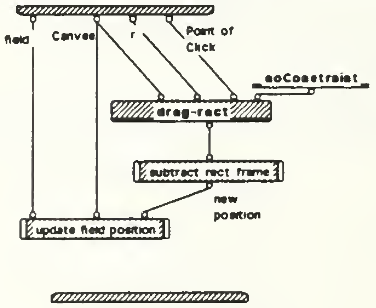
If click fell within rect, stop this local and return rect value

▨IECanvas/IE Click 3:4process simple click 1:2click inside rect? 1:1inside rect? 1:1get rect frame 1:1

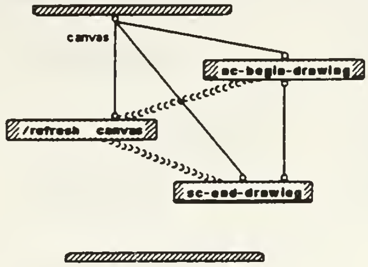


on canvas, rect is framed. add the InsetRect amount to Position to ensure proper rect is presented to PtInRect

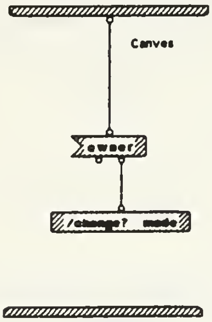
▨IECanvas/IE Click 3:4process simple click 1:2process 2:2do drag 1:1



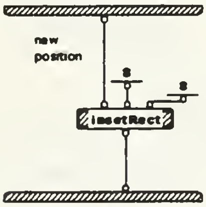
▨IECanvas/IE Click 3:4process simple click 1:2process 2:2do draw 1:1



▨IECanvas/IE Click 3:4process simple click 1:2process 2:2update change? status 1:1

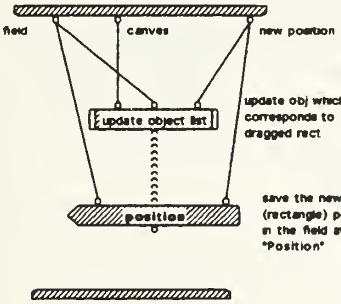


▨IECanvas/IE Click 3:4process simple click 1:2process 2:2do drag 1:1subtract rect frame 1:1



rect was enlarged when
originally drawn to provide
separation between edit text
and rect frame. must remove
this extra area when re-drawing
rect, or else rect will grow

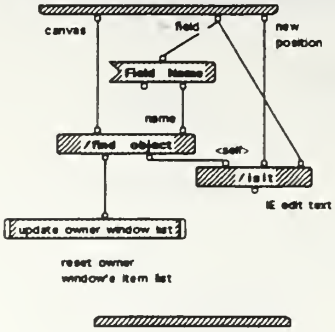
▨IECanvas/IE Click 3:4process simple click 1:2process 2:2do drag 1:1update field position 1:1



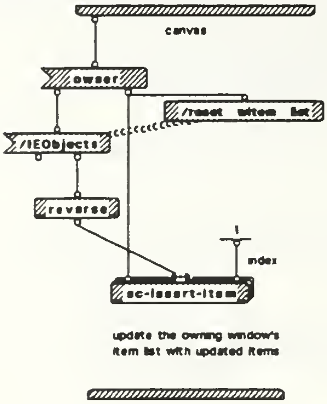
update obj which
corresponds to
dragged rect

save the new field
(rectangle) position
in the field attribute
"Position"

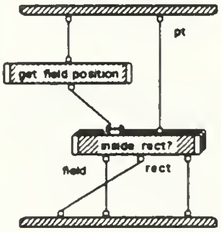
▨IECanvas/IE Click 3:4process simple click 1:2process 2:2do drag 1:1update field position 1:1update object list



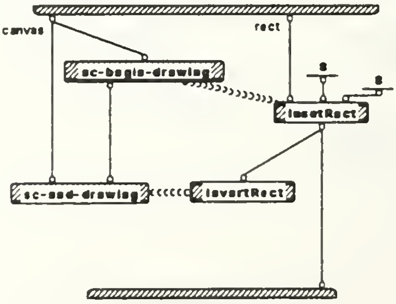
▨IECanvas/IE Click 3:4process simple click 1:2process 2:2do drag 1:1update field position 1:1update object list 1
▨pdate owner window list 1:1



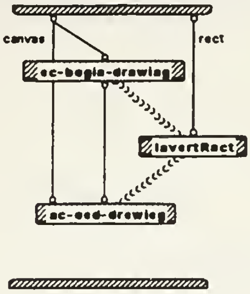
▨IECanvas/IE Click 4:4process double-click 1:2click in rect? 1:1



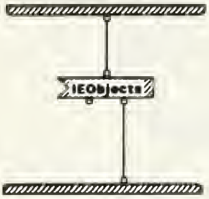
▨IECanvas/IE Click 4:4process double-click 1:2highlight the rect 1:1



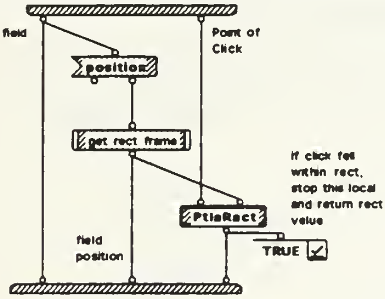
▨IECanvas/IE Click 4:4process double-click 1:2unhighlight the rect 1:1



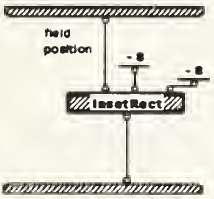
▨IECanvas/IE Click 4:4process double-click 1:2click in rect? 1:1get field position 1:1



▨IECanvas/IE Click 4:4process double-click 1:2click in rect? 1:1inside rect? 1:1

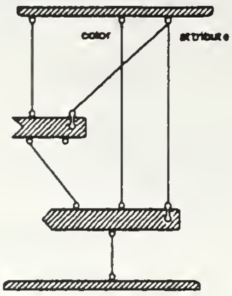


▨IECanvas/IE Click 4:4process double-click 1:2click in rect? 1:1inside rect? 1:1get rect frame 1:1

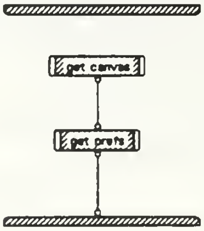


on canvas, rect is framed.
add the InsetRect amount
to Position to ensure proper
rect is presented to PtInRect

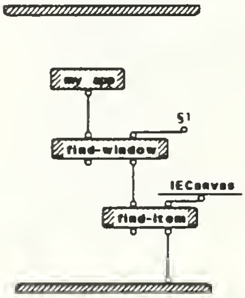
▨IECanvas/set prefs 1:1



▨IECanvas/get prefs 1:1

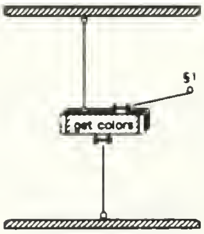


▨IECanvas/get prefs 1:1get canvas 1:1



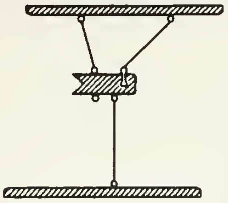
§1. Interface Editor Window

▨IECanvas/get prefs 1:1get prefs 1:1

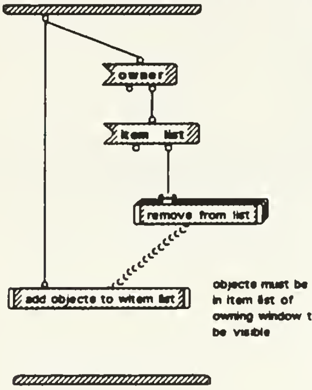


§1 (background foreground)

▨IECanvas/get prefs 1:1get prefs 1:1get colors 1:1

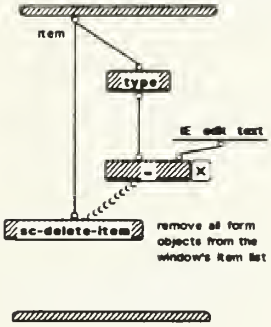


▨IECanvas/update witems 1:1



objects must be in item list of owning window to be visible

▨IECanvas/update witems 1:1remove from list 1:2

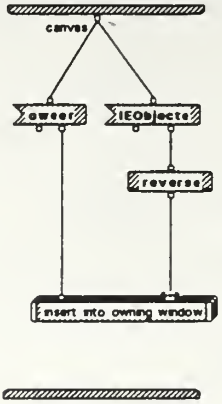


remove all form objects from the window's item list

▨IECanvas/update witems 1:1remove from list 2:2

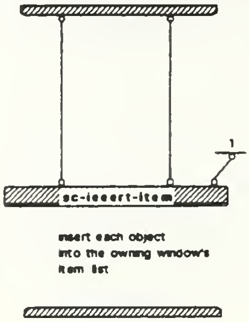
item is not a canvas object. no action required

IECanvas/update witems 1:1add objects to witem list 1:1

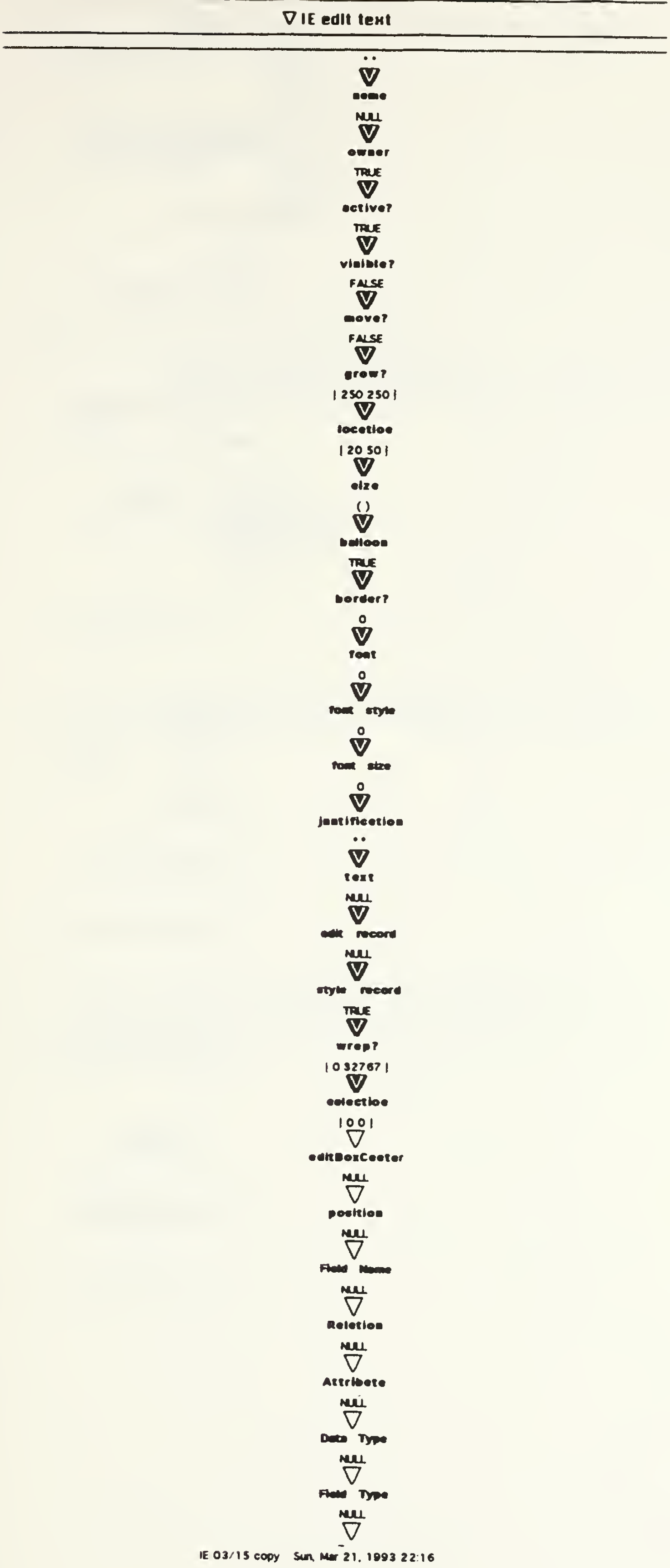


reverse list
to preserve
tab order

IECanvas/update witems 1 1add objects to witem list 1 1insert into owning window 1 1

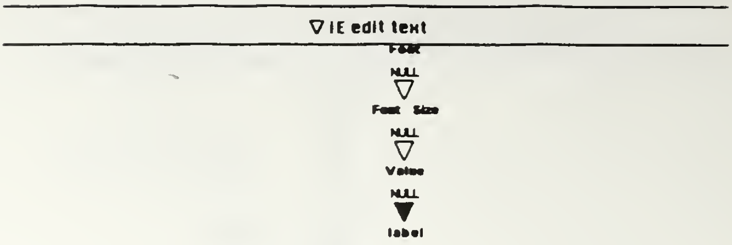


insert each object
into the owning window's
item list

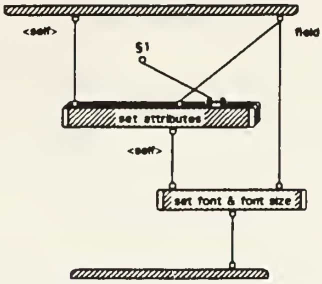


```
∇ IE edit text

""
∇ name
NULL
∇ owner
TRUE
∇ active?
TRUE
∇ visible?
FALSE
∇ move?
FALSE
∇ grow?
( 250 250 )
∇ location
( 20 50 )
∇ size
()
∇ balloon
TRUE
∇ border?
0
∇ font
0
∇ font style
0
∇ font size
0
∇ justification
""
∇ text
NULL
∇ edit record
NULL
∇ style record
TRUE
∇ wrap?
( 0 32767 )
∇ selection
( 0 0 )
∇ editBoxCenter
NULL
∇ position
NULL
∇ Field Name
NULL
∇ Relation
NULL
∇ Attribute
NULL
∇ Data Type
NULL
∇ Field Type
NULL
∇
```

IE 03/15 copy   Sun, Mar 21, 1993 22:16

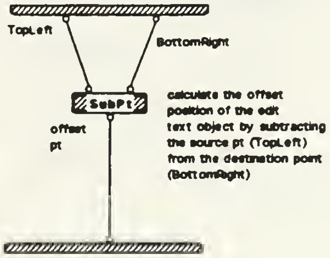

▽ IE edit text
Font
NULL
▽
Font Size
NULL
▽
Value
NULL
▼
label

▨IE edit text/init 1:1xfer field data 1:1



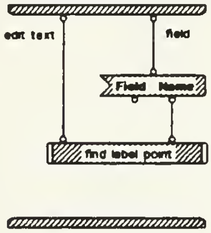
$1 ("Field Name" "Data Type" "Relation" "Attribute" "Field Type" "Font" "Font Size")

▨IE edit text/init 1:1calculate size 1:1

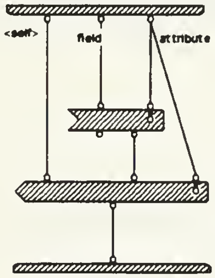


calculate the offset position of the edit text object by subtracting the source pt (TopLeft) from the destination point (BottomRight)

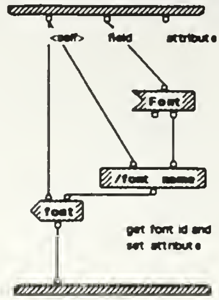
▨IE edit text/init 1:1calculate label 1:1



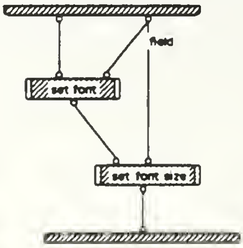
▨IE edit text/init 1:1xfer field data 1:1set attributes 1:2



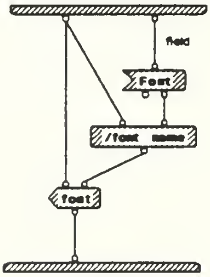
▨IE edit text/init 1:1xfer field data 1:1set attributes 2:2



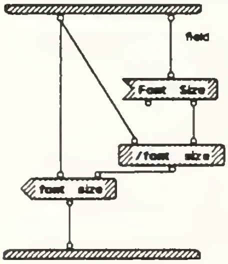
▨IE edit text/init 1:1xfer field data 1:1set font & font size 1:1



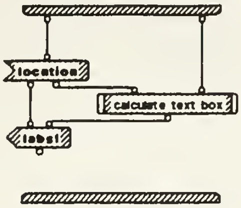
▨IE edit text/init 1:1xfer field data 1:1set font & font size 1:1set font 1:1



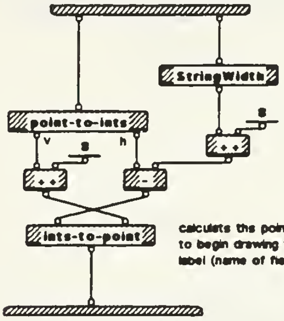
▨IE edit text/init 1:1xfer field data 1:1set font & font size 1:1set font size 1:1



▨IE edit text/init 1:1calculate label 1:1find label point 1:1

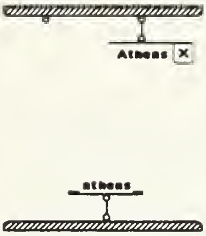


▨IE edit text/init 1:1calculate label 1:1find label point 1:1calculate text box 1:1

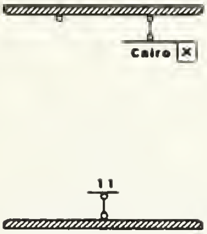


calculats the point at which to begin drawing the field label (name of field)

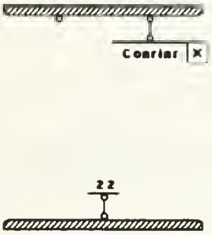
▨IE edit text/font name 1:14

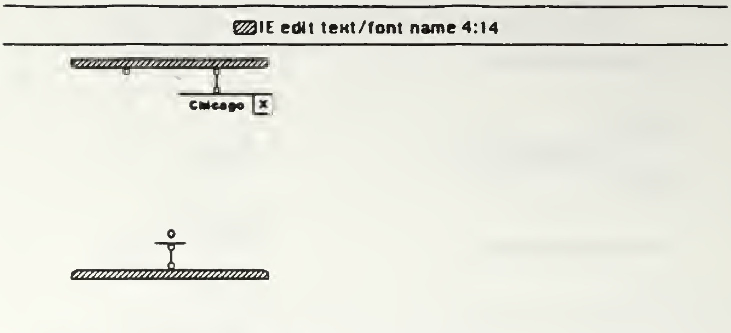


▨IE edit text/font name 2:14

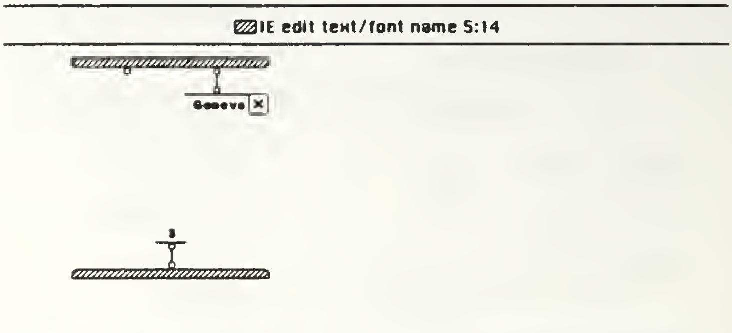


▨IE edit text/font name 3:14

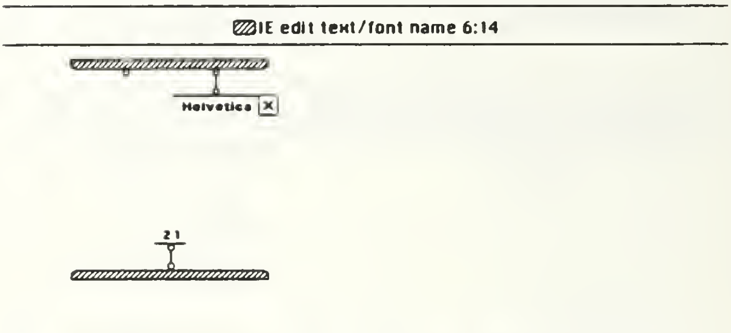




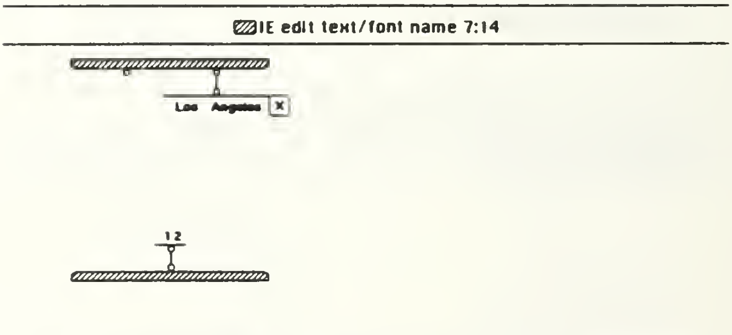
▨IE edit text/font name 4:14



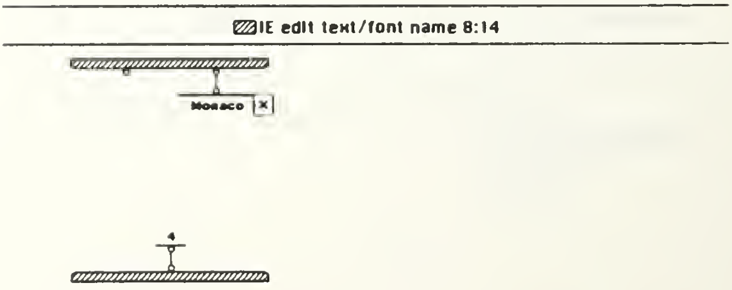
▨IE edit text/font name 5:14



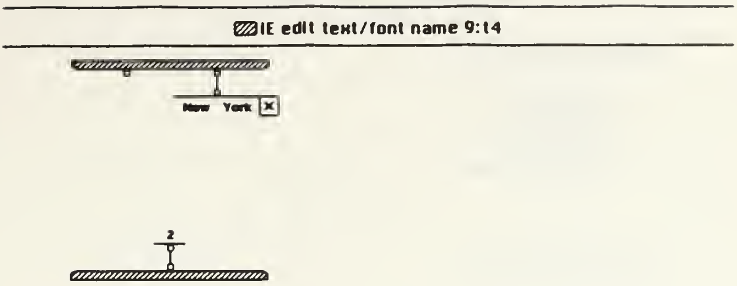
▨IE edit text/font name 6:14

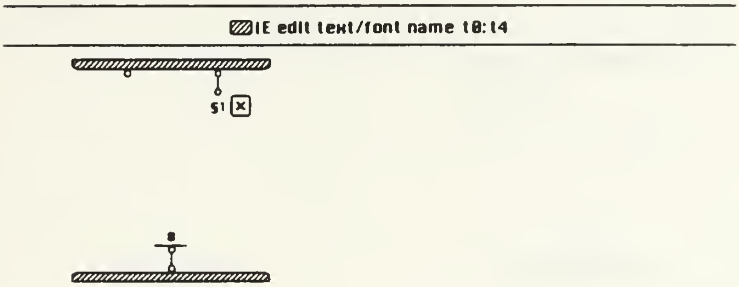


▨IE edit text/font name 7:14

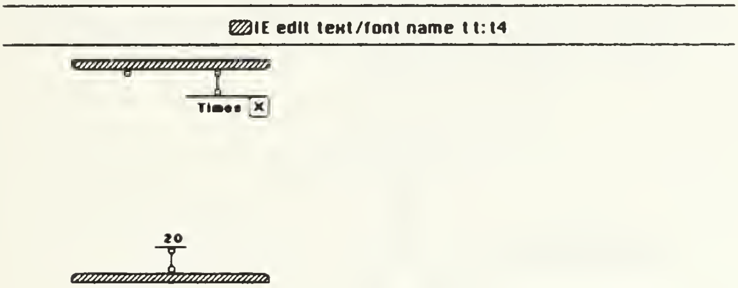


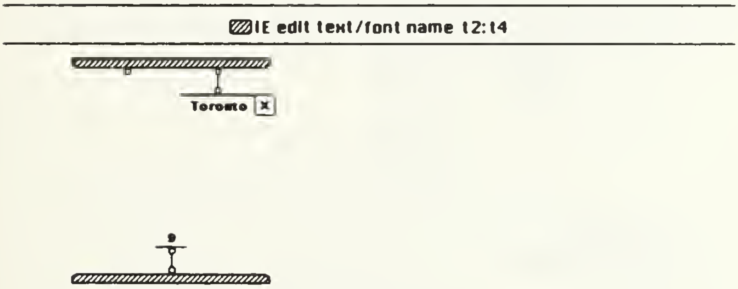
▨IE edit text/font name 8:14

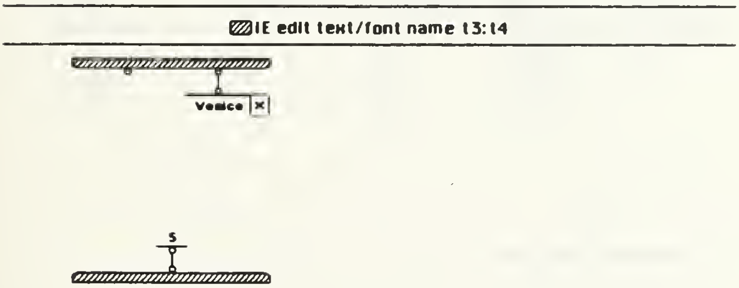




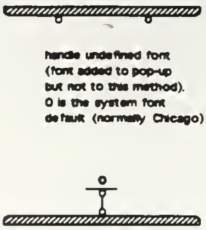
§1 San Francisco





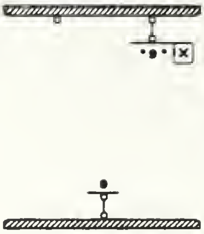


▨IE edit text/font name 14:14

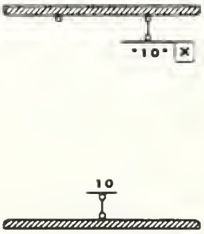


handle undefined font (font added to pop-up but not to this method). 0 is the system font default (normally Chicago)

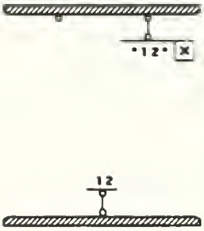
▨IE edit text/font size 1:6



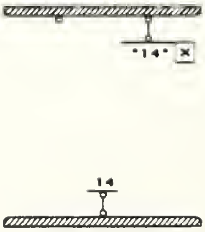
▨IE edit text/font size 2:6



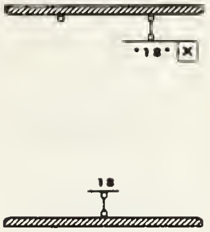
▨IE edit text/font size 3:6



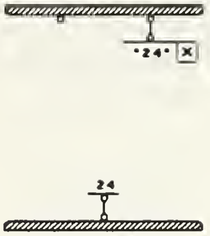
▨IE edit text/font size 4:6



▨IE edit text/font size 5:6



▨IE edit text/font size 6:6



∇Form Menu

"Forms"
∇
name

NULL
∇
owner

FALSE
∇
active?

NULL
∇
menu record

FALSE
∇
keys

FALSE
∇
enabled?

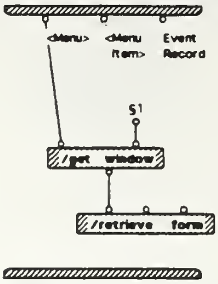
(<<Menu Ite
∇
item list

(1 "Forms W
∇
balloon

Form Menu

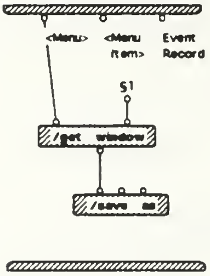
delete form
edit form
save form
open form
get window
retrieve form
new form
close form
save form as

▨Form Menu/open form 1:1



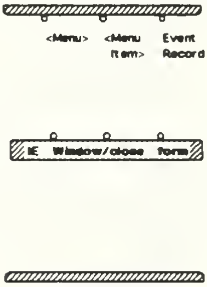
§1 Interface Editor Window

▨Form Menu/save form 1:1

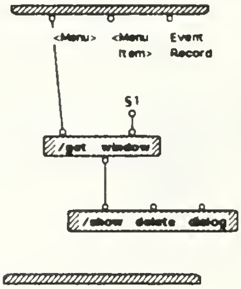


§1 Interface Editor Window

▨Form Menu/close form 1:1

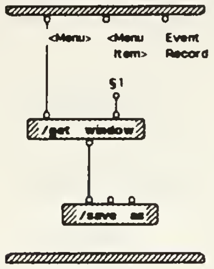


▨Form Menu/delete form 1:1



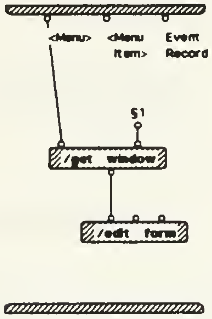
§1. Interface Editor Window

▨Form Menu/save form as 1:1



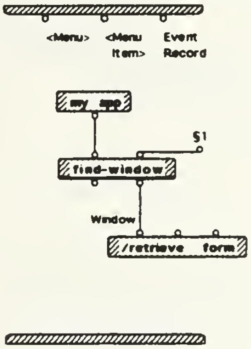
§1 Interface Editor Window

▨Form Menu/edit form 1:1



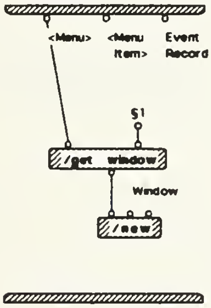
§1 Interface Editor Window

▨Form Menu/retrieve form 1:1



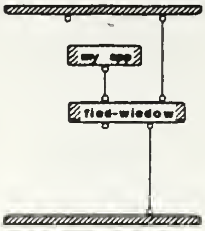
§1 Interface Editor Window

▨Form Menu/new form 1:1



§1 Interface Editor Window

▨Form Menu/get window 1:1



▽Window Menu

"Window"
▽
name

NULL
▽
owner

FALSE
▽
active?

NULL
▽
menu record

FALSE
▽
keys

TRUE
▽
enabled?

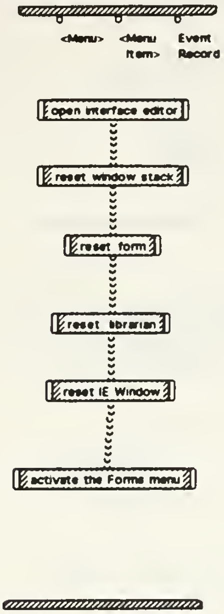
(<<Menu Ite…
▽
item list

(1 "Window …
▽
balloon

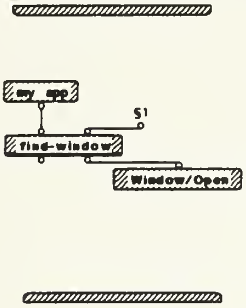
Window Menu

open interface — opens the IE Window and initializes module editor

▨Window Menu/open interface editor 1:1

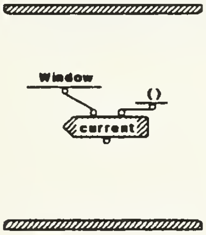


▨Window Menu/open interface editor 1:1open interface editor 1:1

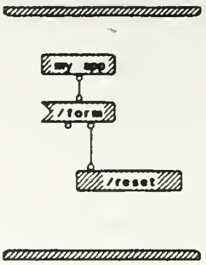


§1 Interface Editor Window

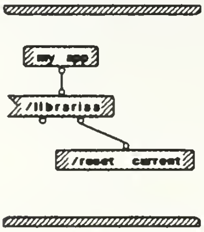
▨Window Menu/open interface editor 1:1reset window stack 1:1



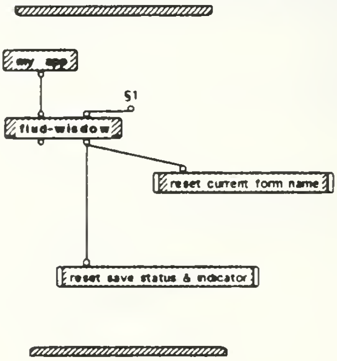
▨Window Menu/open interface editor 1:1reset form 1:1



▨Window Menu/open interface editor 1:1reset librarian 1:1

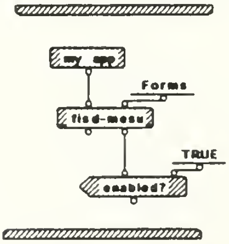


▨Window Menu/open interface editor 1:1reset IE Window 1:1

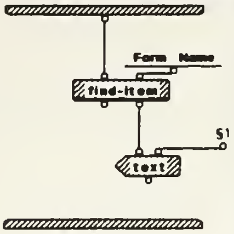


§1 Interface Editor Window

▨Window Menu/open interface editor 1:1activate the Forms menu 1:1

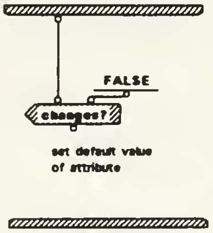


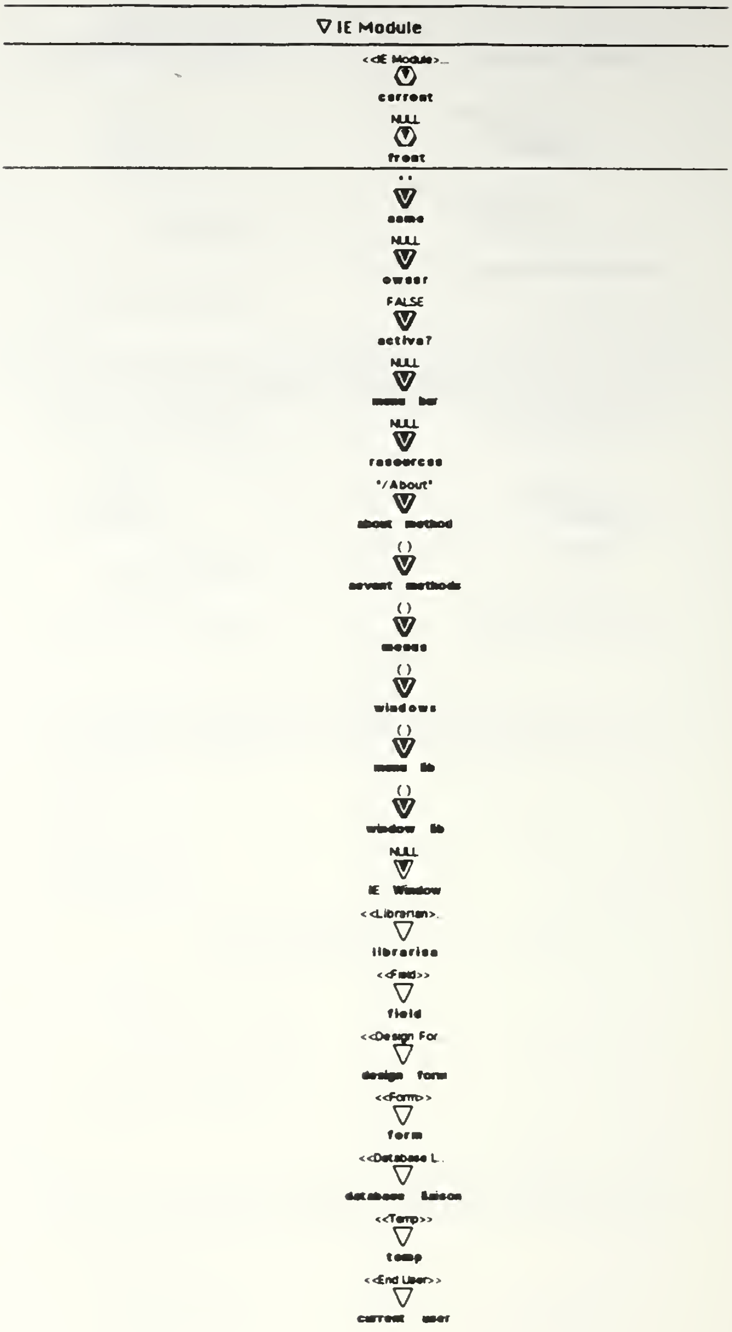
▨ Window Menu/open interface editor 1:1 reset IE Window 1:1 reset current form name 1:1



§1. <no currently active form>

▨ Window Menu/open interface editor 1:1 reset IE Window 1:1 reset save status & indicator 1:1





▽ IE Module

<<IE Module>>...
current

NULL
front

" "
name

NULL
owner

FALSE
active?

NULL
menu bar

NULL
resources

"/About"
about method

()
event methods

()
menus

()
windows

()
menu lib

()
window lib

NULL
IE Window

<<Librarian>>.
libraries

<<Field>>
field

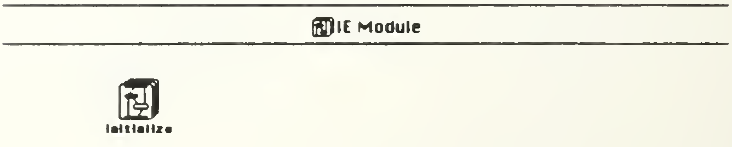
<<Design For
design form

<<Form>>
form

<<Database L.
database liaison

<<Temp>>
temp

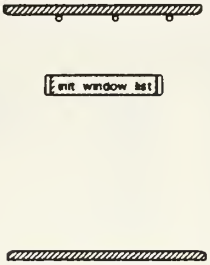
<<End User>>
current user



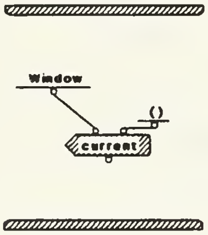
IE Module

initialize

▨IE Module/initialize 1:1

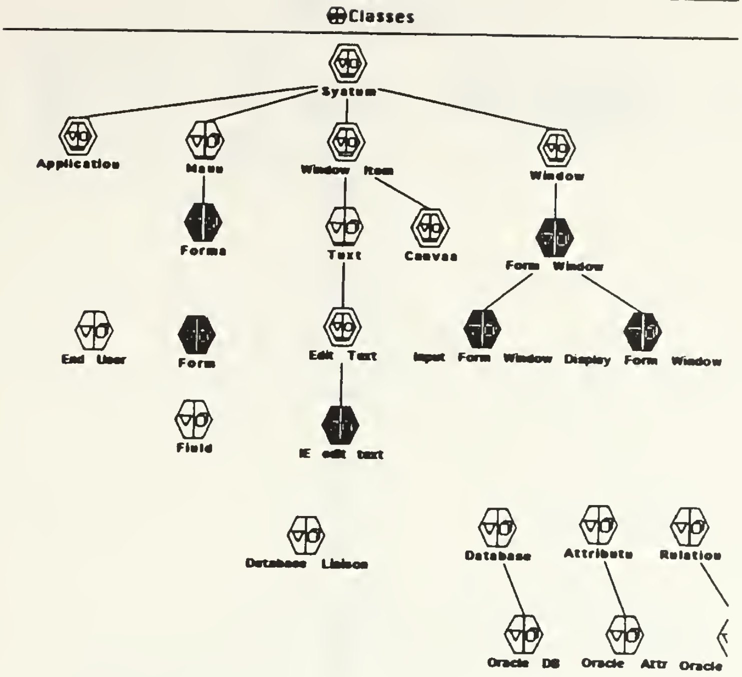


▨IE Module/initialize 1:1init window list 1:1

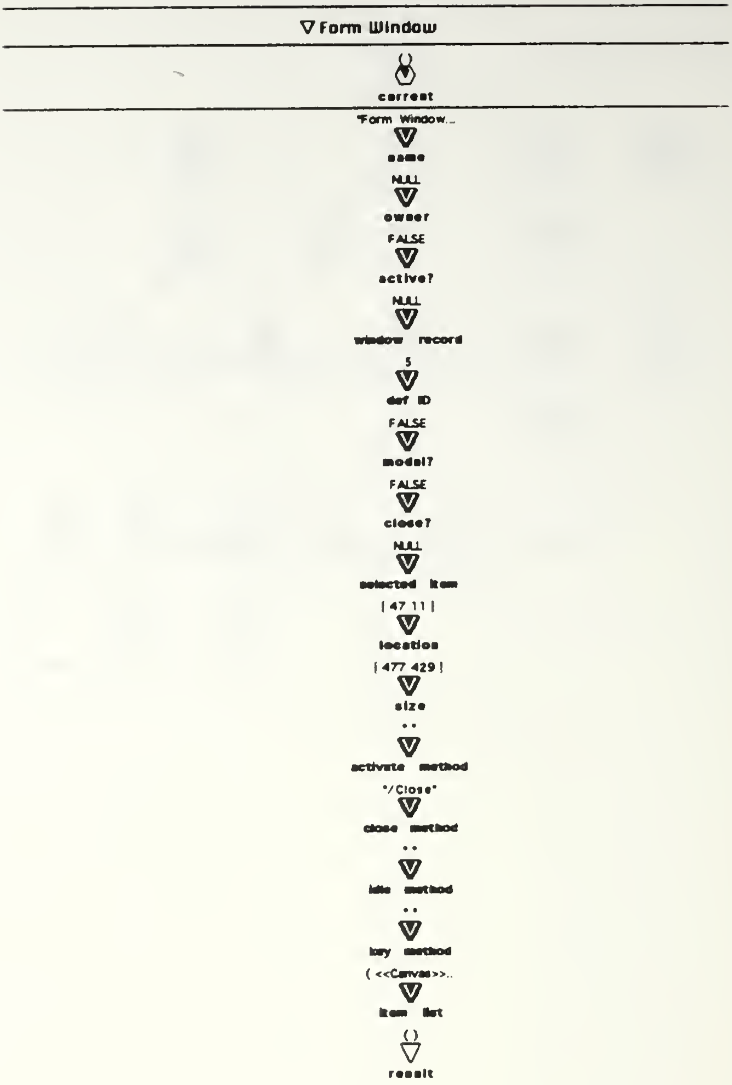


PROGRAM LISTING FOR:

THE FORM USE APPLICATION



IE small 03/15 bak Mon, Mar 22, 1993 8:12



∇ Form Window

current

"Form Window...
name

NULL
owner

FALSE
active?

NULL
window record

5
def ID

FALSE
modal?

FALSE
close?

NULL
selected item

{ 47 11 }
location

{ 477 429 }
size

• •
activate method

"/Close"
close method

• •
idle method

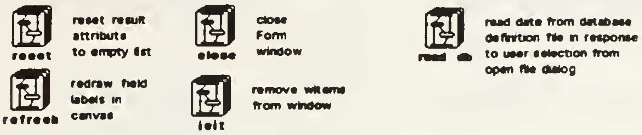
• •
key method

(<<Canvas>>..
item list

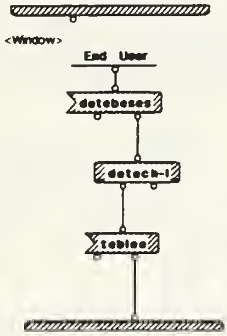
( )
result

IE small 03/15 bak   Mon, Mar 22, 1993 8 12

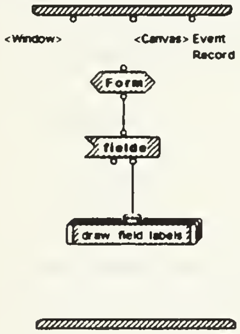
Form Window


reset
reset result attribute to empty list
close
close Form window
read db
read date from database definition file in response to user selection from open file dialog
refresh
redraw field labels in canvas
init
remove witems from window

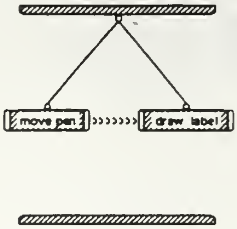
Form Window/read db 1:1



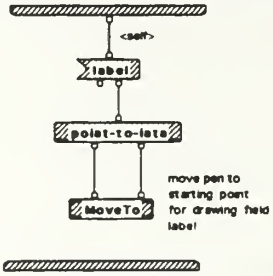
Form Window/refresh 1:1



▨Form Window/refresh 1:1draw field labels 1:1

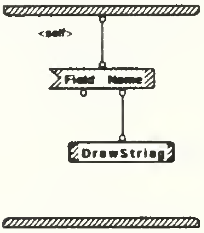


▨Form Window/refresh 1:1draw field labels 1:1move pen 1:1

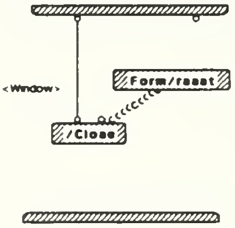


move pen to starting point for drawing field label

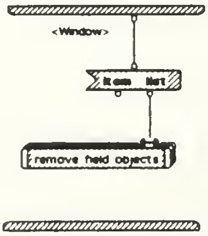
▨Form Window/refresh 1:1draw field labels 1:1draw label 1:1



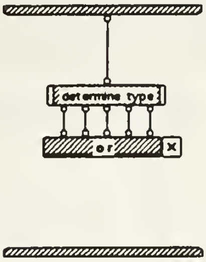
▨Form Window/close 1:1



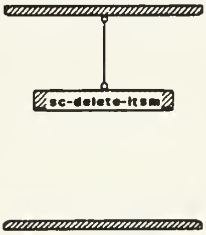
▨Form Window/init 1:1



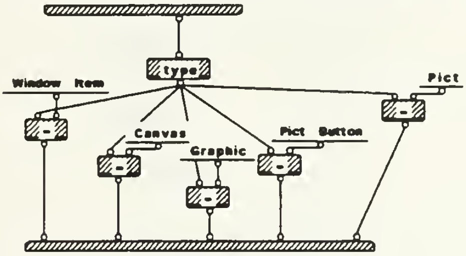
▨Form Window/init 1:1remove field objects 1:2



▨Form Window/init 1:1remove field objects 2:2

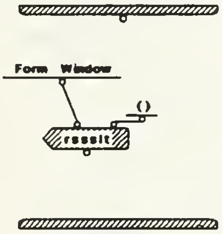


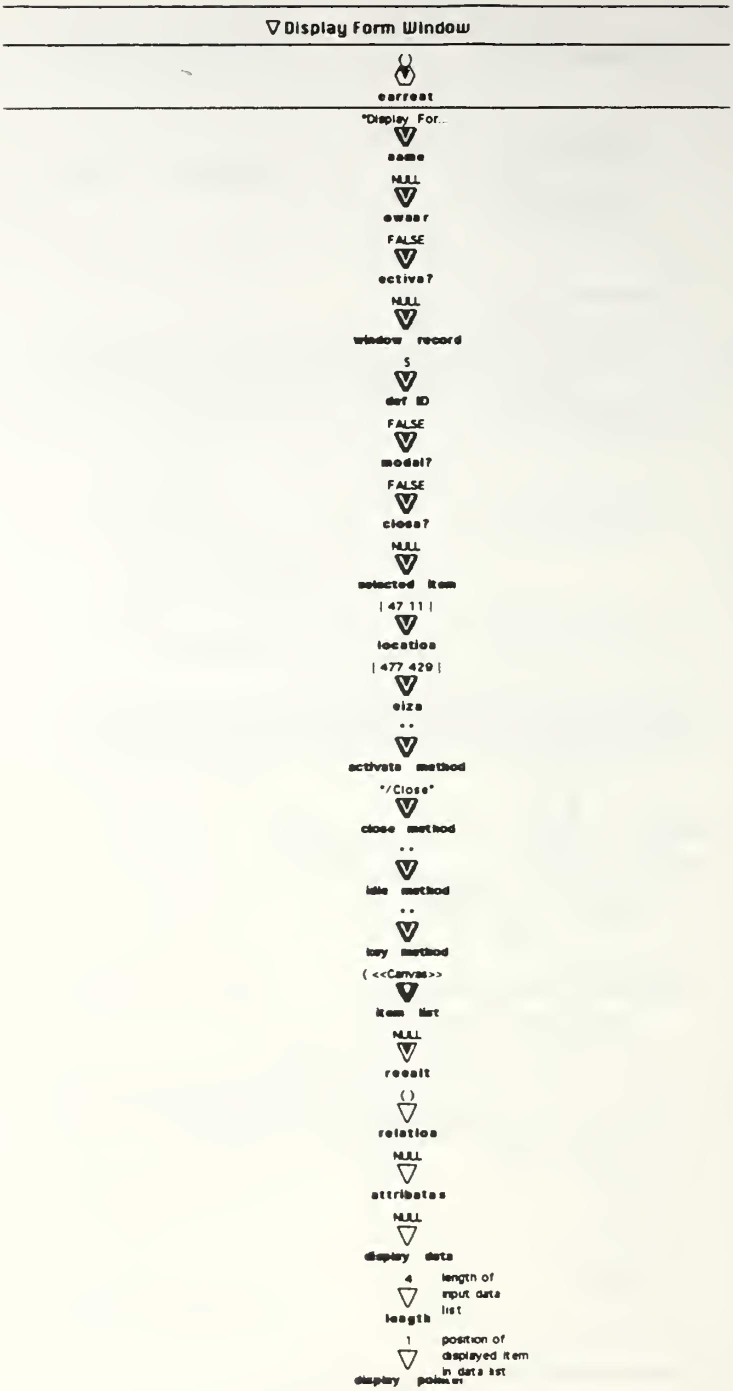
▨Form Window/init 1:1remove field objects 1:2determine type 1:1



these items belong to the window,
and don't need to be removed

▨Form Window/reset 1:1





∇ Display Form Window

current

"Display For...
name

NULL
owner

FALSE
active?

NULL
window record

5
def ID

FALSE
modal?

FALSE
close?

NULL
selected item

| 47 11 |
location

| 477 429 |
size

• •
activate method

"/Close"
close method

• •
idle method

• •
key method

(<<Canvas>>
item list

NULL
result

( )
relation

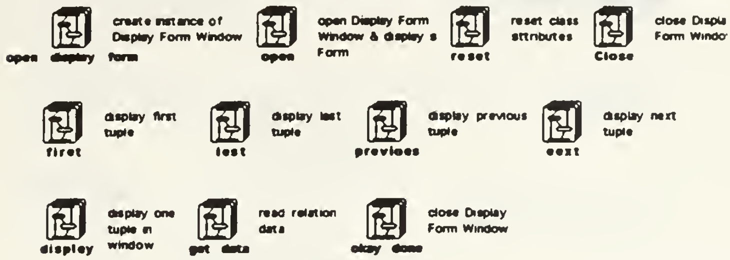
NULL
attributes

NULL
display data

4
length
length of input data list

1
display pointer
position of displayed item in data list

Display Form Window



open display form — create instance of Display Form Window

open — open Display Form Window & display s Form

reset — reset class attributes

Close — close Displa Form Windo

first — display first tuple

last — display last tuple

previous — display previous tuple

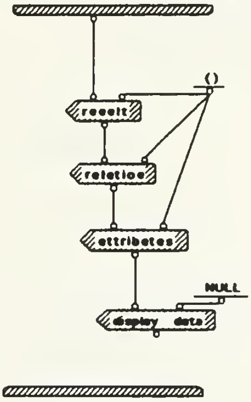
next — display next tuple

display — display one tuple in window

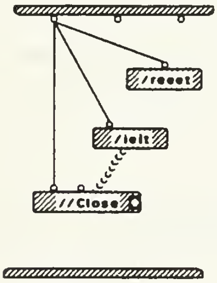
get data — read relation data

okay done — close Display Form Window

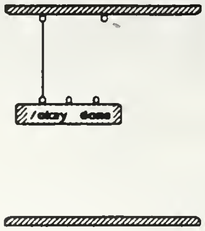
Display Form Window/reset 1:1



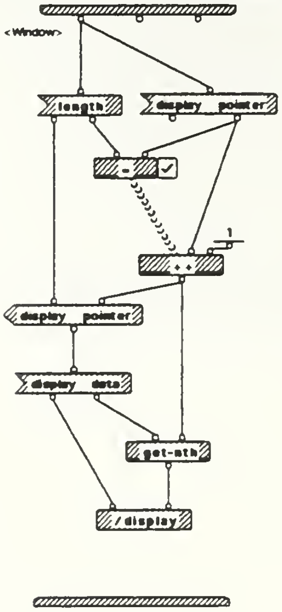
Display Form Window/okay done 1:1



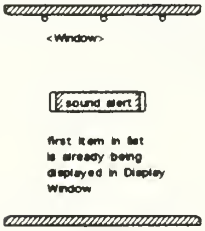
▨Display Form Window/Close 1:1



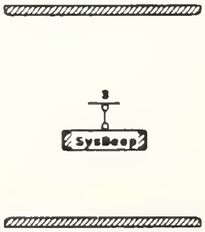
▨Display Form Window/next 1:2



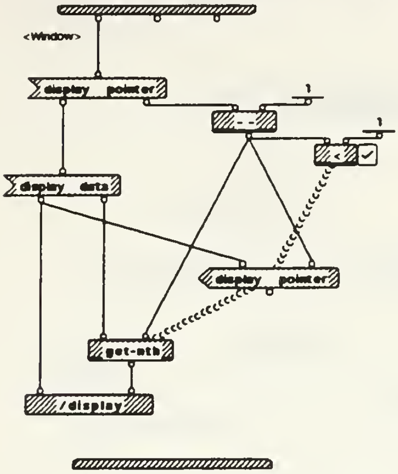
▨Display Form Window/next 2:2



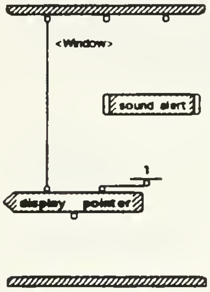
▨Display Form Window/next 2:2sound alert 1:1



▨Display Form Window/previous 1:2

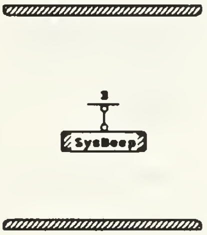


▨Display Form Window/previous 2:2

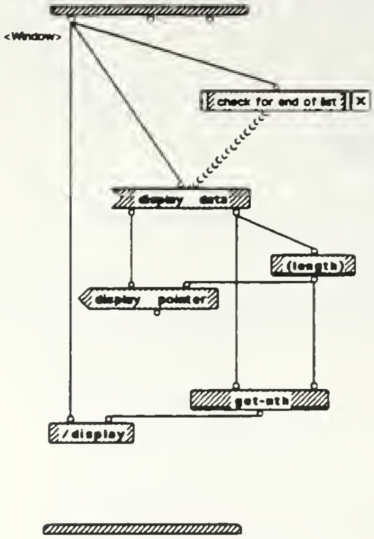


first item in list is already being displayed in Display Window

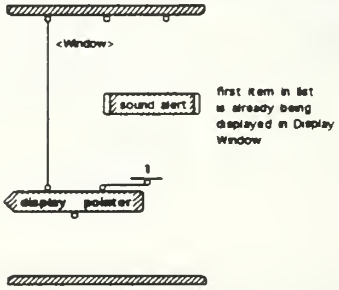
▨Display Form Window/previous 2:2sound alert 1:1



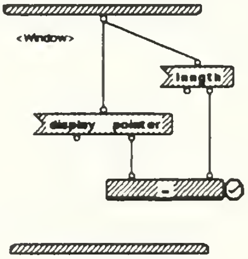
▨Display Form Window/last 1:2



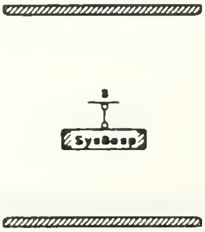
▨Display Form Window/last 2:2



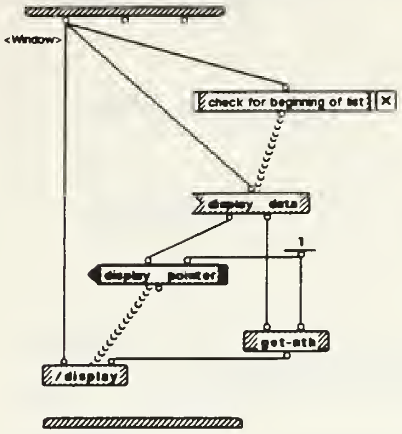
▨Display Form Window/last 1:2check for end of list 1:1



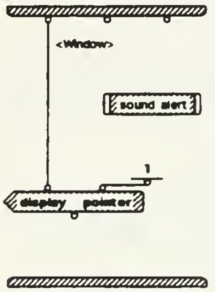
▨Display Form Window/last 2:2sound alert 1:1



▨Display Form Window/first 1:2

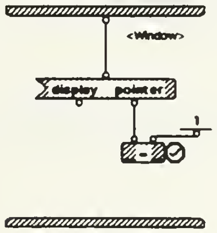


▨Display Form Window/first 2:2

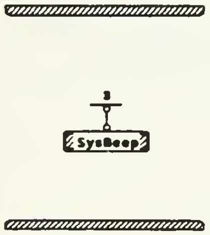


first item in list
is already being
displayed in Display
Window

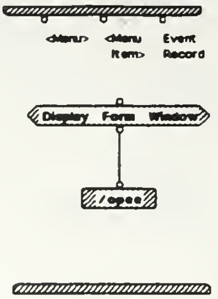
▨Display Form Window/first 1:2check for beginning of list 1:1



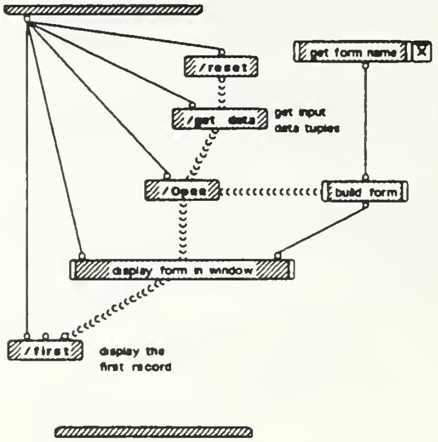
▨Display Form Window/first 2:2sound alert 1:1



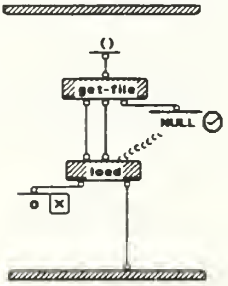
▨Display Form Window/open display form 1:1



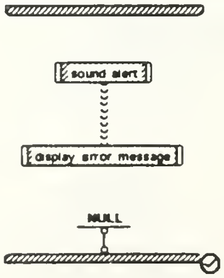
▨Display Form Window/open 1:1



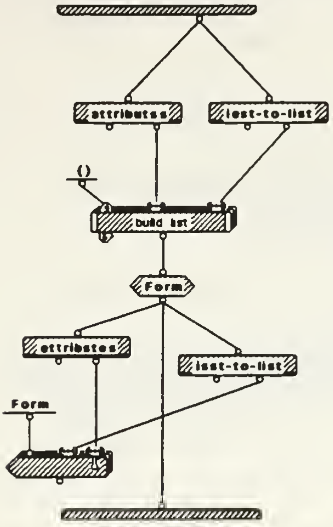
▨Display Form Window/open 1:1get form name 1:2



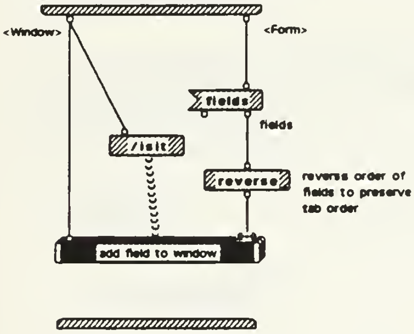
▨Display Form Window/open 1:1get form name 2:2



▨Display Form Window/open 1:1build form 1:1

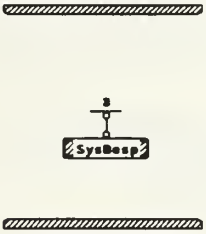


▨Display Form Window/open 1:1display form in window 1:1

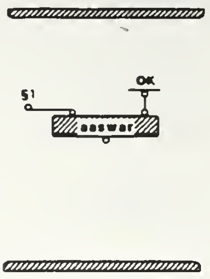


reverse order of fields to preserve tab order

▨Display Form Window/open 1:1get form name 2:2sound alert 1:1

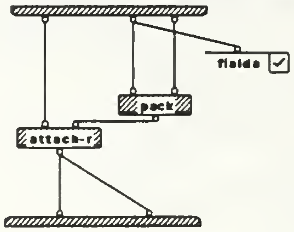


▨Display Form Window/open 1:1get form name 2:2display error message 1:1

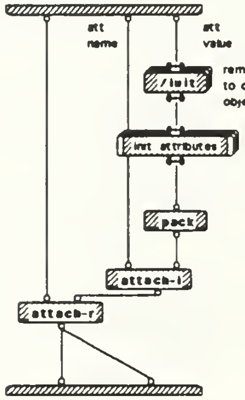


§1 Unable to load form. Please verify form name and location.

▨Display Form Window/open 1:1build form 1:1build list 1:2

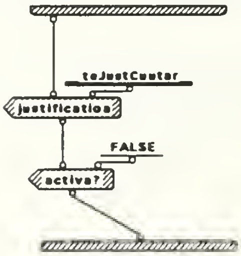


▨Display Form Window/open 1:1build form 1:1build list 2:2

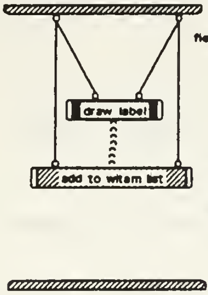


remove all reference
to owning window from
object creation application

▨Display Form Window/open 1:1build form 1:1build list 2:2init attributes 1:1

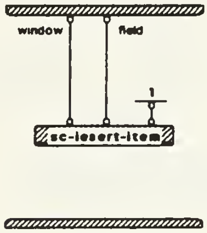


▨Display Form Window/open 1:1 display form in window 1 1 add field to window 1 1

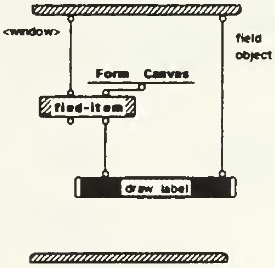


add form objects
to window item list
of input window objects
will not be visible unless
included in window's item
list

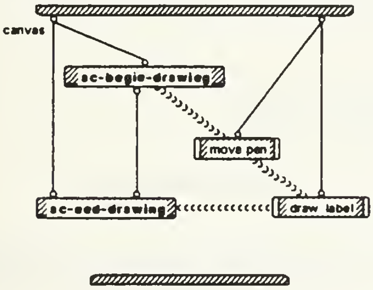
▨Display Form Window/open 1:1 display form in window 1:1 add field to window 1:1 add to witem list 1 1



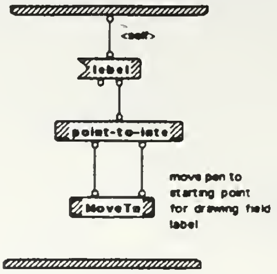
▨Display Form Window/open 1:1 display form in window 1 1 add field to window 1 1 draw label 1 1



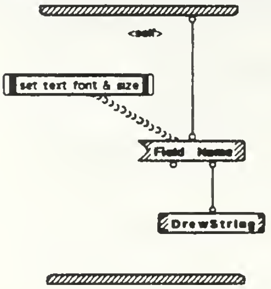
▨Display Form Window/open 1:1 display form in window 1:1 add field to window 1:1 draw label 1:1 draw label 1 1



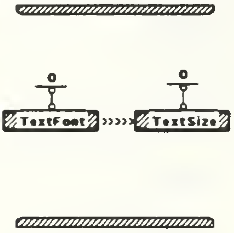
▨Display Form Window/open 1:1 display form in window 1:1 add field to window 1:1 draw label 1:1 draw label 1:1 move pen 1:1



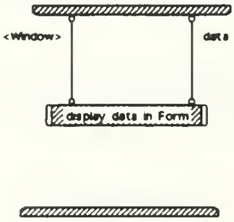
▨Display Form Window/open 1:1 display form in window 1:1 add field to window 1:1 draw label 1:1 draw label 1:1 draw label 1:1



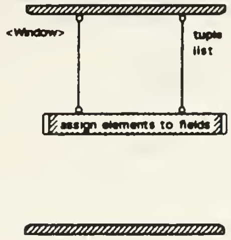
▨Display Form Window/open 1:1 display form in window 1:1 add field to window 1:1 draw label 1:1 draw label 1:1 draw label 1:1 set text font & size 1:1



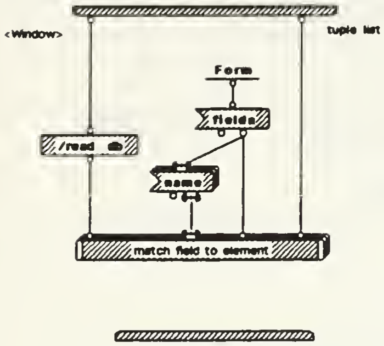
▨Display Form Window/display 1:1



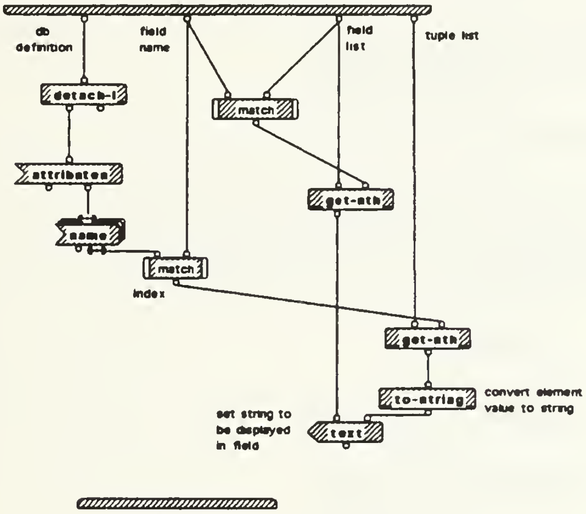
▨Display Form Window/display 1:1display data in Form 1:1



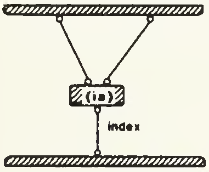
▨Display Form Window/display 1:1display date in Form 1:1assign elements to fields 1:1



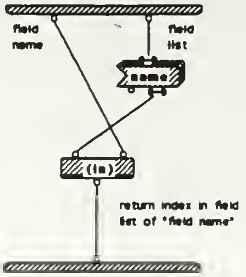
▨Display Form Window/display 1:1display data in Form 1:1assign elements to fields 1.1match field to element 1 1



▨Display Form Window/display 1:1display date in Form 1.1assign elements to fields 1 1match field to element 1 1n
▨tch 1.1

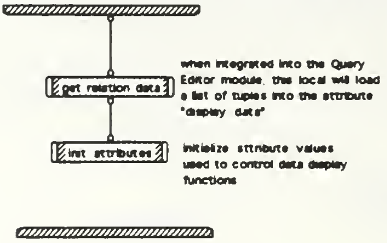


▨Display Form Window/display 1:1display data in Form 1:1assign elements to fields 1 1match field to element 1.1n ▨tch 1:1



return index in field list of "field name"

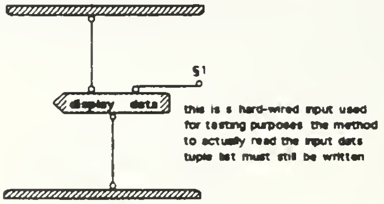
▨Display Form Window/get data 1:1



when integrated into the Query Editor module, the local will load a list of tuples into the attribute "display data"

initialize sttribute values used to control data display functions

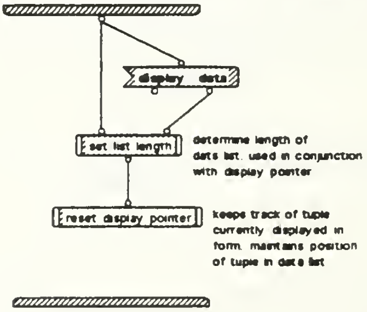
▨Display Form Window/get data 1:1get relation data 1:1



this is s hard-wired input used for testing purposes. the method to actually read the input data tuple list must still be written

§1 ((1 2 3 4 5 6) (one two three four five six) (I II III IV V VI) (oden dva tree chiten pyatz shest) (i ii iii iv v vi))

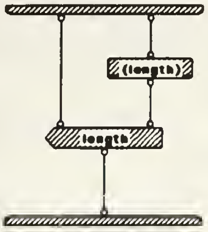
▨Display Form Window/get data 1:1init attributes 1:1



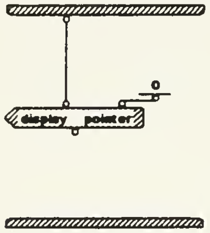
determine length of data list. used in conjunction with display pointer

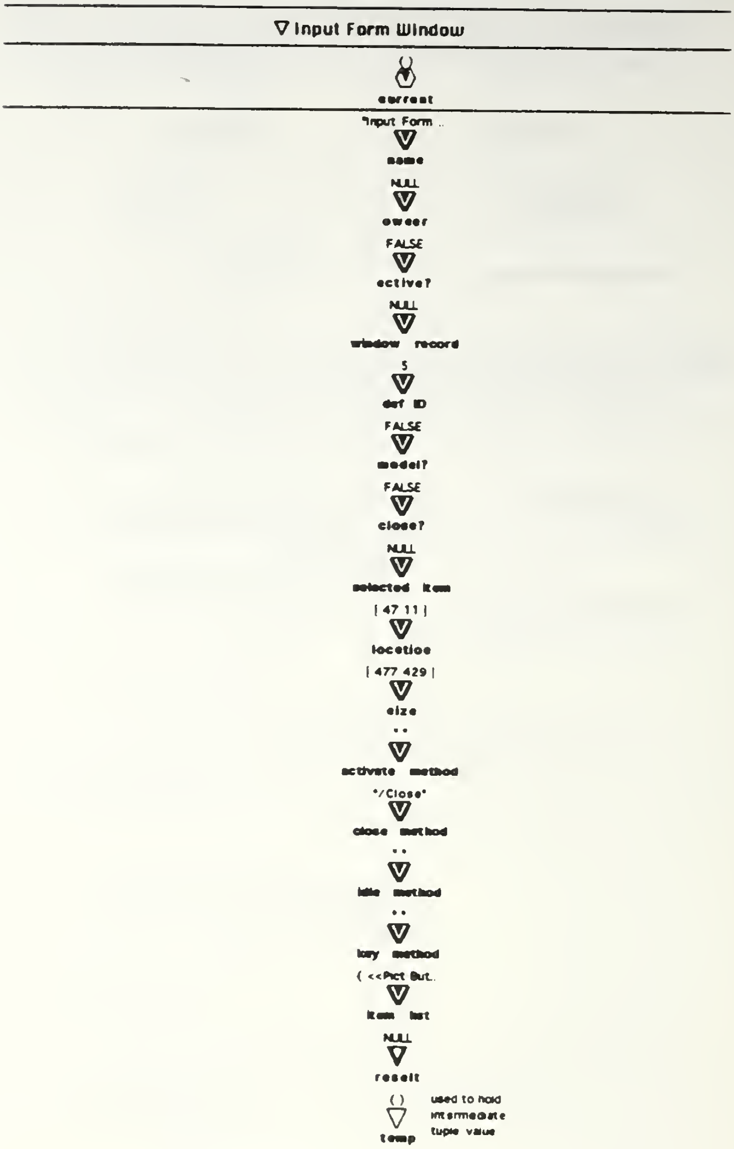
keeps track of tuple currently displayed in form. maintains position of tuple in data list

▨Display Form Window/get data 1:1init attributes 1:1set list length 1:1



▨Display Form Window/get data 1:1init attributes 1:1reset display pointer 1:1





∇ Input Form Window

current

"Input Form ..
name

NULL
owner

FALSE
active?

NULL
window record

5
def ID

FALSE
modal?

FALSE
close?

NULL
selected item

[47 11]
location

[477 429]
size

• •
activate method

"/Close"
close method

• •
idle method

• •
key method

(<<Pict But..
item list

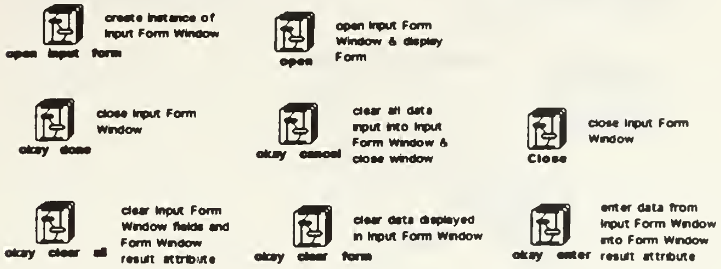
NULL
result

( )
temp

used to hold
intermediate
tuple value

IE small 03/15.bak Mon, Mar 22 1993 8:12

Input Form Window



open input form — create instance of Input Form Window

open — open Input Form Window & display Form

okay done — close Input Form Window

okay cancel — clear all data input into Input Form Window & close window

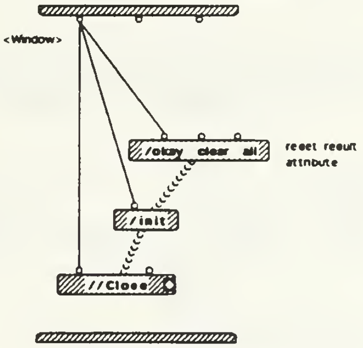
Close — close Input Form Window

okay clear all — clear Input Form Window fields and Form Window result attribute

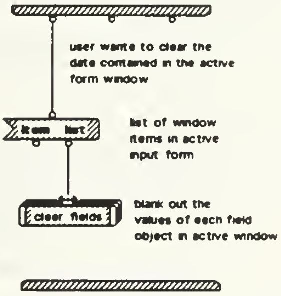
okay clear form — clear data displayed in Input Form Window

okay enter — enter data from Input Form Window into Form Window result attribute

Input Form Window/okay cancel 1:1

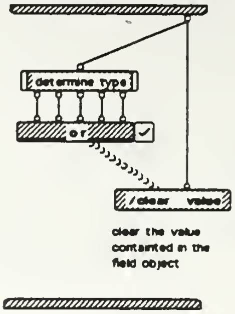


Input Form Window/okay clear form 1:1



user wants to clear the data contained in the active form window
item list
list of window items in active input form
clear fields
blank out the values of each field object in active window

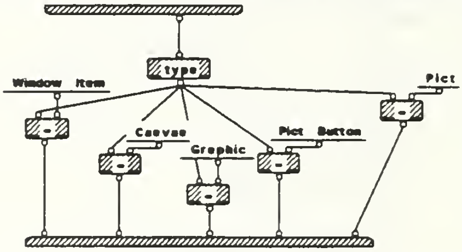
▨Input Form Window/okay clear form 1:1clear fields 1:2



clear the value contained in the field object

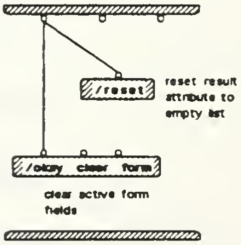
▨Input Form Window/okay clear form 1:1clear fields 2:2

▨Input Form Window/okay clear form 1:1clear fields 1:2determine type 1:1



these items belong to the window, and don't need to be removed

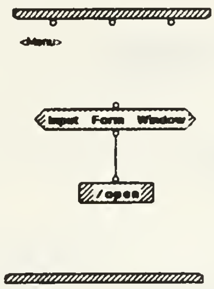
▨Input Form Window/okay clear all 1:1



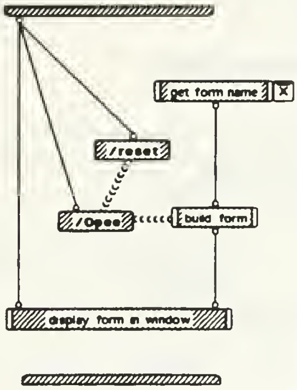
reset result attribute to empty list

clear active form fields

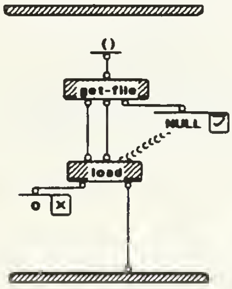
▨Input Form Window/open input form 1:1



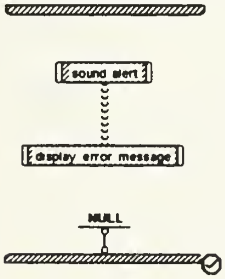
▨Input Form Window/open 1:1



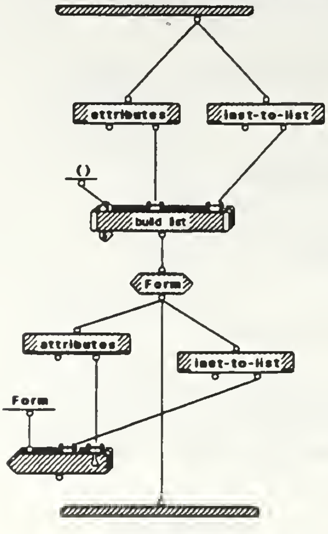
▨Input Form Window/open 1:1get form name 1:2



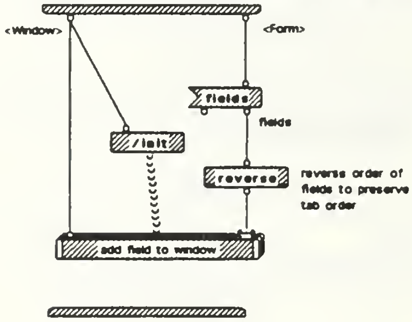
▨Input Form Window/open 1:1get form name 2:2



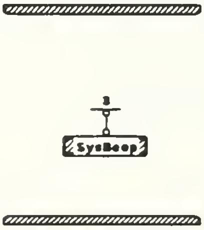
▨Input Form Window/open 1:1build form 1:1



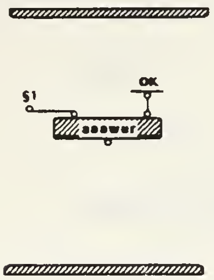
▨Input Form Window/open 1:1display form in window 1:1



▨Input Form Window/open 1:1get form name 2:2sound alert 1:1

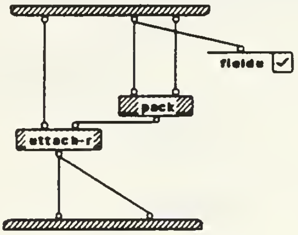


▨Input Form Window/open 1:1get form name 2:2display error message 1:1

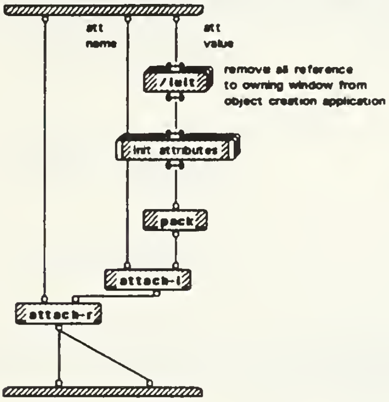


§1 Unable to load form. Please verify form name and location.

▨Input Form Window/open 1:1build form 1:1build list 1:2

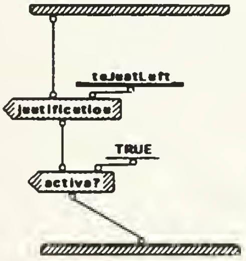


▨Input Form Window/open 1:1build form 1:1build list 2:2

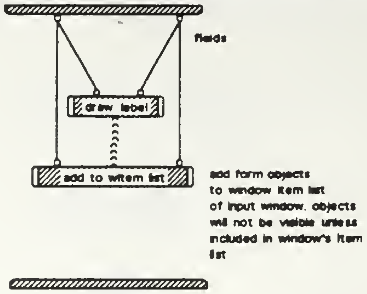


remove all reference
to owning window from
object creation application

▨Input Form Window/open 1:1build form 1:1build list 2:2init attributes 1:1

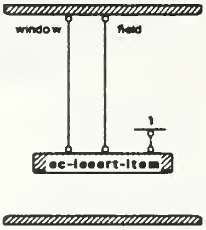


▨Input Form Window/open 1:1 display form in window 1:1 add field to window 1:1

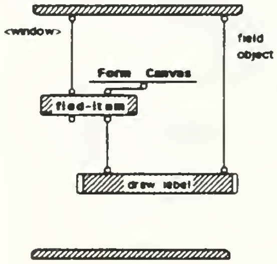


add form objects
to window item list
of input window. objects
will not be visible unless
included in window's item
list

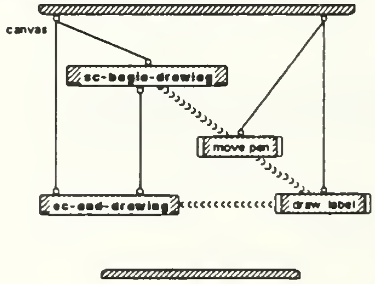
▨Input Form Window/open 1 1 display form in window 1:1 add field to window 1.1 add to witem list 1.1



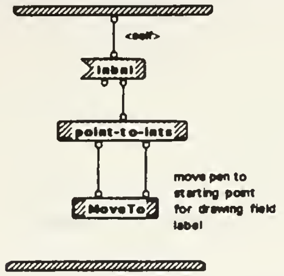
▨Input Form Window/open 1 1 display form in window 1.1 add field to window 1:1 draw label 1.1



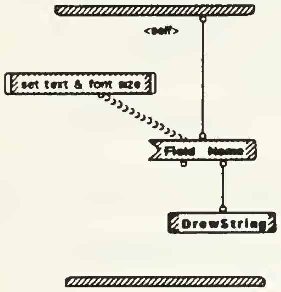
▨Input Form Window/open 1:1 display form in window 1 1 sdd field to window 1:1 draw label 1:1 draw label 1:1



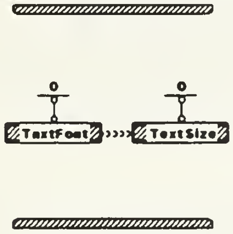
▨Input Form Window/open 1:1display form in window 1:1add field to window 1:1draw label 1:1draw label 1:1move
▨en 1:1



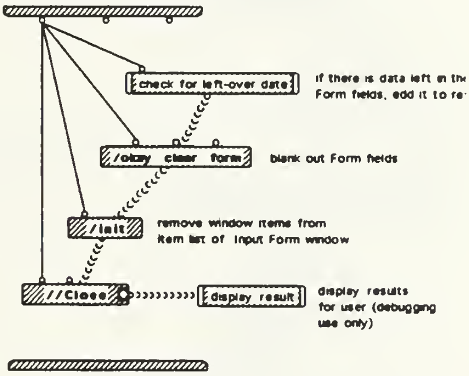
▨Input Form Window/open 1:1display form in window 1:1add field to window 1:1draw label 1:1draw label 1:1draw l
▨abel 1:1



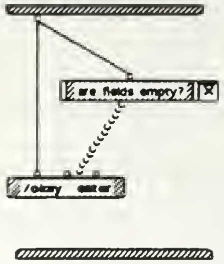
▨Input Form Window/open 1:1display form in window 1:1add field to window 1:1draw label 1:1draw label 1:1draw l
▨abel 1:1set text & font size 1:1



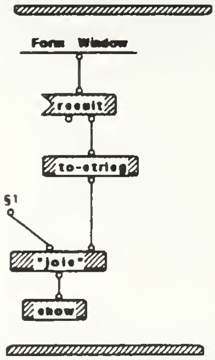
▨Input Form Window/okay done 1:1


check for left-over data
if there is data left in the Form fields, add it to re
/okay clear form
blank out Form fields
/init
remove window items from item list of Input Form window
//Close
display result
display results for user (debugging use only)

▨Input Form Window/okay done 1:1check for left-over data 1:1

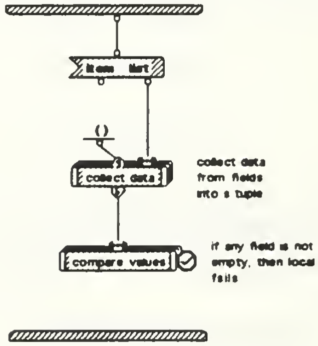


▨Input Form Window/okay done 1:1display result 1:1

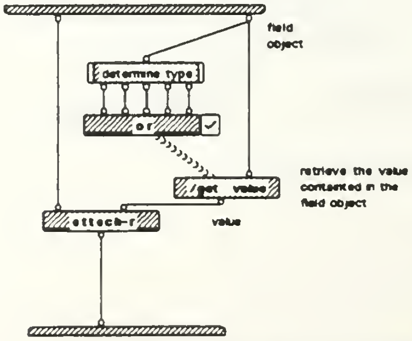


§1 contents of attribute "result" of class "Form Window" is

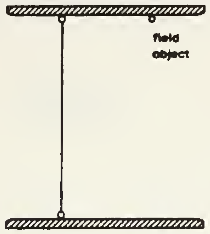
▨Input Form Window/okay done 1:1check for left-over data 1:1are fields empty? 1:1



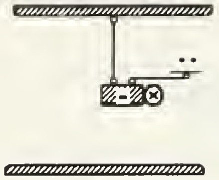
▨Input Form Window/okay done 1 1check for left-over data 1 1are fields empty? 1:1collect data 1 2



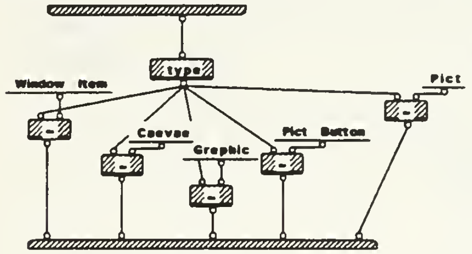
▨Input Form Window/okay done 1:1check for left-over data 1:1are fields empty? 1 1collect data 2.2



▨Input Form Window/okay done 1 1check for left-over data 1:1are fields empty? 1:1compare values 1 1

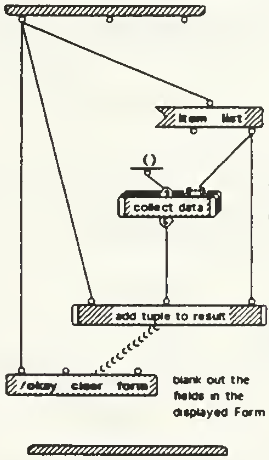


▨Input Form Window/okay done 1:1check for left-over data 1.1are fields empty? 1 1collect data 1:2determine type
▨1·1



these items belong to the window,
and don't need to be removed

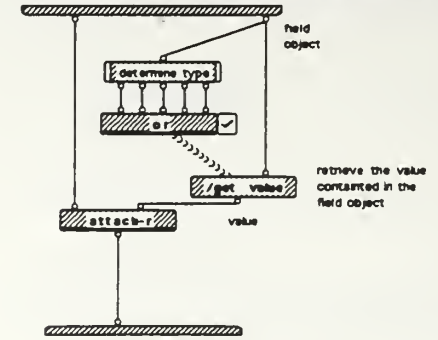
▨Input Form Window/okay enter 1:1



collect
from f
into a t

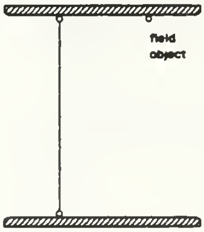
blank out the
fields in the
displayed Form

▨ Input Form Window/okay enter 1:1collect data 1:2

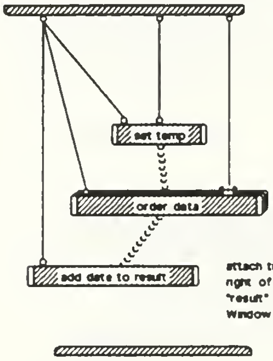


retrieve the value contained in the field object

▨ Input Form Window/okay enter 1:1collect data 2:2



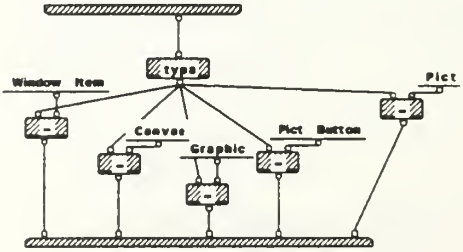
▨ Input Form Window/okay enter 1:1add tuple to result 1:1



arrange tuple elements according to relation definition

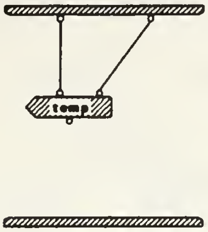
attach tuple to right of attribute "result" in class Form Window

▨ Input Form Window/okay enter 1:1collect data 1:2determine type 1:1

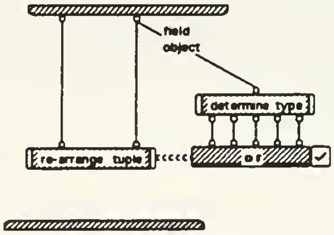


these items belong to the window, and don't need to be removed

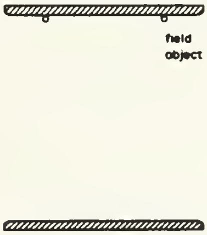
▨Input Form Window/okay enter 1:1add tuple to result 1:1set temp 1:1



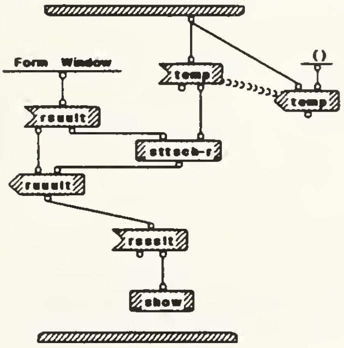
▨Input Form Window/okay enter 1:1add tuple to result 1:1order data 1:2



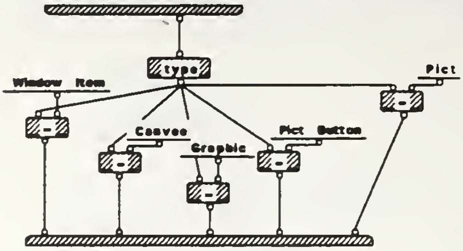
▨Input Form Window/okay enter 1:1add tuple to result 1:1order data 2:2



▨Input Form Window/okay enter 1:1add tuple to result 1:1add data to result 1:1

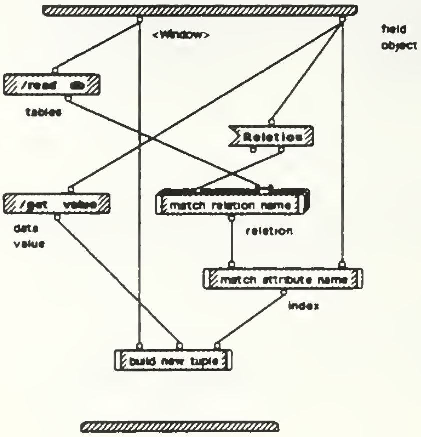


▨Input Form Window/okay enter 1:1add tuple to reeult 1:1order date 1:2determine type 1:1

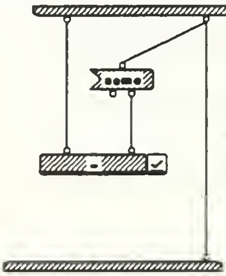


these items belong to the window,
and don't need to be removed

▨Input Form Window/okay enter 1:1add tuple to result 1:1order data 1:2re-arrange tuple 1:1

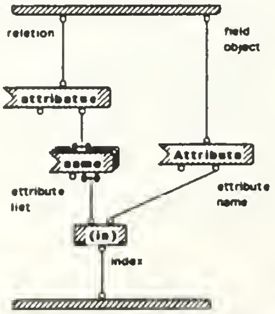


▨Input Form Window/okay enter 1:1add tuple to reeult 1:1order data 1:2re-arrange tuple 1:1match relation neme 1
▨1

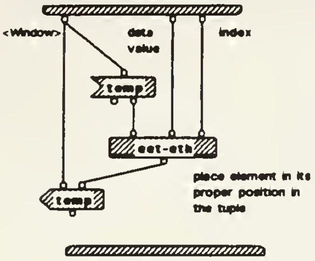


if e match is found,
the name will be
output on the
case output bar
terminel

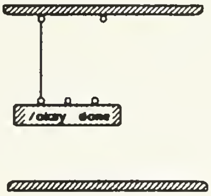
▨Input Form Window/okay enter 1:1add tuple to reeult 1:1order dete 1:2re-arrange tuple 1:1match ettribute name
▨ 1

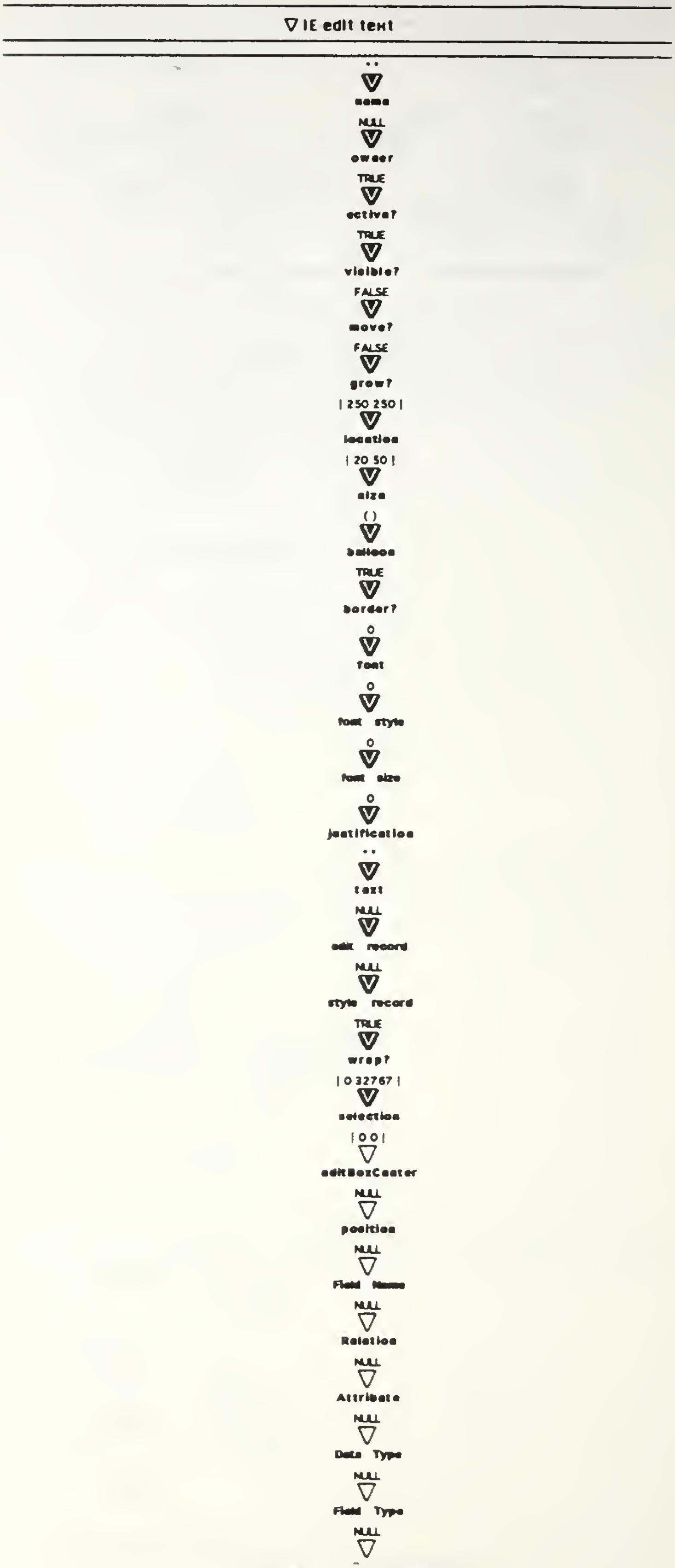


▨Input Form Window/okay enter 1:1 add tuple to result 1:1 order data 1:2 re-arrange tuple 1:1 build new tuple 1 1



▨Input Form Window/Close 1:1





▽ IE edit text

```
• •
▽
name
NULL
▽
owner
TRUE
▽
active?
TRUE
▽
visible?
FALSE
▽
move?
FALSE
▽
grow?
| 250 250 |
▽
location
| 20 50 |
▽
size
( )
▽
balloon
TRUE
▽
border?
0
▽
font
0
▽
font style
0
▽
font size
0
▽
justification
• •
▽
text
NULL
▽
edit record
NULL
▽
style record
TRUE
▽
wrap?
| 0 32767 |
▽
selection
| 0 0 |
▽
editBoxCenter
NULL
▽
position
NULL
▽
Field Name
NULL
▽
Relation
NULL
▽
Attribute
NULL
▽
Data Type
NULL
▽
Field Type
NULL
▽
-
```

∇ IE edit text

Font

NULL

Font Size

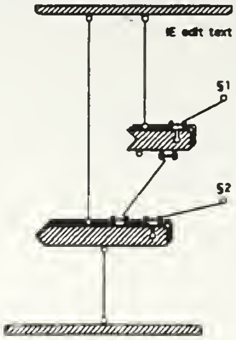
NULL

Value

NULL

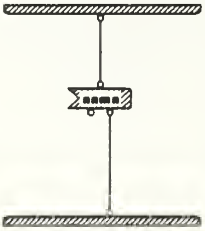
label

▨IE edit text/init 1:1copy attributes 1:1

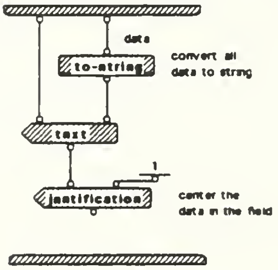


S1 ("name" "location" "size" "font" "Font Size" "position" "Field Name" "Relation" "Attribute" "Data Type" "Field Type
" "label")
S2 (name location size font "Font Size" position "Field Name" Relation Attribute "Data Type" "Field Type" label)

▨IE edit text/name 1:1



▨IE edit text/assign value 1:1



▽Form

"fname ssn"
▽
name

(<<IE edit .
▼
fields

• •
▽
end user

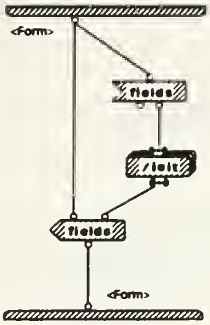
FALSE
▽
protected?

Form

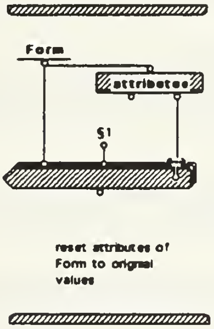
init

reset — reset attributes of Form to orignal values

Form/init 1:1



Form/reset 1:1



§1. (name fields and user "protected?")

∇ Forms

"Forms"
∇
name

NULL
∇
owner

FALSE
∇
active?

NULL
∇
menu record

FALSE
∇
keys

TRUE
∇
enabled?

(<<Menu Ite..
∇
item list

(1 "File Me..
∇
balloon

REFERENCES

[Ambler89] Ambler, Allen, and Burnett, Margaret, "Influence of Visual Technology on the Evolution of Language Environments", *Computer*, October 1989, pages 19-32.

[Apple85] Apple Computer, Inc., *Inside Macintosh, Volumes 1-5*, Addison-Wesley Publishing Company, Inc., 1985.

[Butler90] Butler, Keith, "Collaboration for Technology Transfer - or How Do So Many Promising Ideas Get Lost?", *Conference Proceedings, CHI '90*, ACM Press, 1990.

[Chang87] Chang, S.K., "Visual Languages: A Tutorial and Survey", *IEEE Software*, January 1987, pages 7-17.

[Clark91] Clark, Gard, and Wu, C.T., *Dataflow Query Language for Relational Databases*, Department of Computer Science, Naval Postgraduate School, Monterey, CA.

[Codd88a] Codd, E.F., "Fatal Flaws in SQL: Part I", *Datamation*, v. 34, August 1988.

[Codd88b] Codd, E.F., "Fatal Flaws in SQL: Part II", *Datamation*, v. 34, September 1988.

[Codd90] Codd, E.F., *The Relational Model for Database Management: Version 2*, Addison-Wesley, 1990.

[Cox86] Cox, Brad, *Object Oriented Programming - An Evolutionary Approach*, Reading: Addison-Wesley Publishing Company, 1986.

[Dertouzos90] Dertouzos, Michael, "Redefining Tomorrow's User Interface", Plenary Address, *Conference Proceedings, CHI '90*, ACM Press, 1990.

[Elmasri89] Elmasri, R. and Navathe, S., *Fundamentals of Database Systems*, Benjamin/ Cummings Publishing Company, Inc., 1989.

[Erickson90] Erickson, Thomas, Creativity and Design, Introduction to *The Art of Human-Computer Interface Design,* Reading: Addison-Wesley Publishing Company, 1990

[Falby91] Falby, John, Lecture Notes in Database Systems, Naval Postgraduate School, 1991.

[Gibson90] Gibson, Katie, *Form Systems*, Department of Computer Science, Oregon State University, 1990

[Grudin90] Grudin, Jonathan, "The Computer Reaches Out: The Historical Continuity of Interface Design", *Conference Proceedings, CHI '90,* ACM Press, 1990, pages 261-268.

[Laurel90] Laurel, Brenda, editor, *The Art of Human-Computer Interface Design*. Reading: Addison-Wesley Publishing Company, 1990.

[Locke] Locke, John, "Using UNIX in the Computer Science Department", undated tutorial, Department of Computer Science, Naval Postgraduate School, Monterey, CA.

[MacLennan87] MacLennan, Bruce, *Principles of Programming Languages (second edition))*, Holt, Rinehart & Winston, Inc., 1987.

[Mountford90] Mountford, Joy, et al, "Designers: Meet your Users" (Panel), *CHI '90 Conference Proceedings*, ACM Press, 1990, pages 439-442

[Nielsen90a] Nielsen, Jakob, and Molich, Rolf, "Heuristic Evaluation of User Interfaces", *Conference Proceedings, CHI '90*, ACM Press, 1990, pages 249 - 256.

[Nielsen90b] Nielsen, Jakob, *Hypertext and Hypermedia*. San Diego: Academic Press, 1990.

[Russell92] Russell, Matthew, Xu, Howard and Wang, Lingtao, "Action Assignable Graphics - A Flexible Human-Computer Interface Design Process", Panel Discussion, *Conference Proceedings, CHI '92,* ACM Press, 1992

[Shneiderman92] Shneiderman, Ben, *Designing the User Interface - Strategies for Effective Human-Computer Interaction*, Addison Wesley Publishing Company, Inc., 1992.

[Smith91] Smith, David, *Concepts of Object-Oriented Programming*, McGraw Hill, Inc., 1991.

[Symantec91] Symantec Corporation, *Think C Object-Oriented Programming Manual*, 1989-1991

[TGS90a] The Gunakara Sun Systems, *Prograph 2.0 Technical Specifications*, 1990.

[TGS90b] The Gunakara Sun Systems, *Prograph Tutorial*, 1990.

[TGS90c] The Gunakara Sun Systems, *Prograph Reference*, 1990.

[TGS91] The Gunakara Sun Systems, *Prograph 2.5 Updates*, 1991.

[Tognazzini91] Tognazzini, Bruce, *Tog on Interface*, Addison Wesley Publishing Company, Inc., 1991.

[Winograd90] Winograd, Terry, "What Can We Teach About Human-Computer Interaction", closing address for CHI '90, *Conference Proceedings, CHI '90*, ACM Press, 1990.

[Wu91a] Wu, C. Thomas, *Visual Query Language for Relational Database Interoperability*, Department of Computer Science, Naval Postgraduate School, Monterey, CA., 1991

[Wu91b] Wu, C. Thomas, *Object-Oriented Programming (Class Notes - cs4114)*, Department of Computer Science, Naval Postgraduate School, Monterey, CA., 1991

[Wu91c] Wu, C. Thomas, "OOP + Visual Dataflow Diagram = Prograph", *Journal of Object-Oriented Programming*, June, 1991, pages 71-75.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library 2
Code 52
Naval Postgraduate School
Monterey, CA 93943-5002

Chairman, Code CS 2
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5002

Professor C. Thomas Wu, Code CS/Wu 1
Naval Postgraduate School
Monterey, CA 93943-5002

LCDR John A. Daley, USN, Code CS/Da 1
Naval Postgraduate School
Monterey, CA 93943-5002

Commander, Naval Security Group Command
Code G30 1
3801 Nebraska Ave., NW
Washington, D.C. 20393-5449

LCDR James P. Hargrove, USN 1
148 Kerns Court
Napa, CA 94558