NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

RAPID PRODUCTION OF GRAPHICAL USER INTERFACES

by

David Maurice King
and
Richard Montgomery Prevatt III

December 1990

Thesis Advisor: Michael J. Zyda

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	
ERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS/ZK	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
ADDRESS (City, State, and ZIP Code): Monterey, 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 NO.

TITLE (Include Security Classification)
RAPID PRODUCTION OF GRAPHICAL USER INTERFACES (unclassified)

PERSONAL AUTHOR(S)
King, David Maurice, and Prevatt, Richard Montgomery, III

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 04/89 TO 12/90	December 1990	221

SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the authors and do not reflect the official policy or position 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)
ELD	GROUP	SUB-GROUP	Graphics, Graphical User Interface, User Interface, Simulation, Computer Aided Software Generation, DoD Software Development

ABSTRACT (Continue on reverse if necessary and identify by block number)
There is a growing demand within the military for effective, flexible and configurable command and control rkstations suiting the diversity of experience and working style that commanders bring to the decision making cess. This need motivates development of real-time three-dimensional simulators at the Naval Postgraduate hool. Our work concentrates on the graphical user interface and presents a study of information display, interface man factors, and underlying implementation efficiency considerations so as to enhance real-time simulation tems with minimal degradation in performance.

High quality interface software is costly in time and money, and it is essential for effective system performance. r research culminated in the implementation of the NPS Panel Designer and ToolBox (NPSPD), an automated velopment environment that enables design, implementation, modification, and testing of customized graphical r interfaces. NPSPD includes automatic generation of compilable source code which can stand alone or be grated quickly into a developer's application. NPSPD was developed using Silicon Graphics Inc. IRIS 4D/70GT 4D/GTX workstations, relatively low-cost systems which are commercially available. Methodology used and hniques developed provide a foundation applicable to any hardware capable of a windowing and graphics display.

DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL: Michael J. Zyda	22b. TELEPHONE (Include Area Code): (408) 646-2305 / 22c. OFFICE SYMBOL: CS/ZK

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited

RAPID PRODUCTION OF GRAPHICAL USER INTERFACES

by
David Maurice King
Lieutenant, United States Navy
B.S., North Dakota State University, 1984

and

Richard Montgomery Prevatt III
Lieutenant Commander, United States Navy
B.S.E., Duke University, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1990

ABSTRACT

There is a growing demand within the military for effective, flexible and configurable command and control workstations suiting the diversity of experience and working style that commanders bring to the decision making process. This need motivates development of real-time three-dimensional simulators at the Naval Postgraduate School. Our work concentrates on the graphical user interface and presents a study of information display, interface human factors, and underlying implementation efficiency considerations so as to enhance real-time simulation systems with minimal degradation in performance.

High quality interface software is costly in time and money, and it is essential for effective system performance. Our research culminated in the implementation of the NPS Panel Designer and ToolBox (NPSPD), an automated development environment that enables design, implementation, modification, and testing of customized graphical user interfaces. NPSPD includes automatic generation of compilable source code which can stand alone or be integrated quickly into a developer's application. NPSPD was developed using Silicon Graphics Inc. IRIS 4D/70GT and 4D/GTX workstations, relatively low-cost systems which are commercially available. Methodology used and techniques developed provide a foundation applicable to any hardware capable of a windowing environment and graphics display.

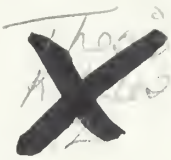

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

NPS Panel Designer and ToolBox was in all respects a team effort by both authors, each contributing to the details of the others work. Lieutenant King developed the panel, actuator, and color editing modules and designed the intermediate file format including the lexical analyzer and parser central to saving and recalling panel design files. His major focus was the development of the Panel Designer interface control, information prompts and the help system. Lieutenant Commander Prevatt designed the data structures and functions used in the Panel Designer and ToolBox, designed the attributes of panels and actuators, implemented the processing and drawing routines, and developed the source code generation module. He provided the overall program design including the format for the generated source code and the means to link it to users' applications.

During our design and development of the NPS Panel Designer and ToolBox, we depended on the support and advice of many individuals. Their assistance made this project possible. We would like to thank the following people for their contributions.

David A. Tristram provided a significant and foundational contribution of the concepts and techniques within the NASA Panel Library, without which this project would not have been conceived. His work provided needed answers to tough design questions.

Lieutenant Commander John M. Yurchak provided his expertise in the tactical use of fleet command and control stations, his amazing knowledge of the C programming language and the UNIX operating system, and his proven and effective approach to software development. Many of the concepts implemented in this design tool have their origin in discussions with John.

Commander Rachel Griffin provided her clear grasp of technical yet readable English to keep this thesis reasonable and her thorough understanding of diverse programming languages to broaden the design considerations.

Lieutenant Andy Anderson, Commander Tom Jurewicz, and Lieutenant John Lyon, bravely chose to use the Panel Designer and ToolBox during its development and before its completion as part of their thesis research and development. Their ALPHA testing of the

Designer software revealed many elusive bugs and identified improvements to overall design that have been incorporated into the final version.

Rosaleen and Tina patiently endured and faithfully encouraged our work. Matthew provide an often needed and refreshing diversion.

Most significantly, we thank our principle advisor, Dr. Michael Zyda, who provided vision, critical advice and guidance during the year comprising this project's development.

I. INTRODUCTION

The continually growing need within the military for effective, flexible and configurable command and control workstations motivates research into real-time information presentation. One fruitful area of this research explores development of real-time three-dimensional simulators having advanced user interfaces. During the past six years, several such simulators have been implemented on Silicon Graphics Inc. (SGI) workstations at the Naval Postgraduate School (NPS), Monterey, California. NPS simulator applications span a diversity of tasks including flight control, ground-based vehicle control, and surface and subsurface ship control.

The complexity of the tactical environment and the three-dimensional and real-time nature of NPS simulators increases the need for advanced user interfaces. The effectiveness of an entire system depends extensively on the interface, its ability to transform data into information, and its ability to clearly and simply provide the user a means to control system operation (Smith and Mosier, 1986, p.296). And as emphasized by Smith, information rather than merely data must be presented.

> When we examine the process of man-computer communication from the human point of view, it is useful to make explicit a distinction which might be described as contrasting "information" with "data". Used in this sense, information can be regarded as the answer to a question, whereas data are the raw materials from which information is extracted.
>
> What the computer can actually provide the man are displays of data. What information he is able to extract from those displays is indicated by his responses. How effectively the data are processed, organized, and arranged prior to presentation will determine how effectively he can and will extract the information he requires from his display. Too frequently these two terms data and information are confused, and the statement, "I need more information," is assumed to mean, "I want more symbols." The reason for the statement, usually, is that the required information is not being extracted from the data. Unless the confusion between data and information is removed, attempts to increase information in a display are directed at obtaining more data, and the trouble is exaggerated rather than relieved. (Smith, 1963, pp.296-297)

A. USER INTERFACE DESIGN

Development of a successful user interface for any application is an expensive and time consuming process, and often the final product does not lend itself to easy modification. One study estimates that user interface design comprises an average of 30 to 35 percent of the time used to produce operational software (Smith and Mosier, 1984). The users' view of a system is "conditioned chiefly by experience with its interface. If the user interface is unsatisfactory, the users' view of the system will be negative" (Smith and Mosier, 1986, p.4).

Recent hardware and software developments significantly affect design and implementation of user interface software. Powerful workstations with bitmapped screens and pointing devices provide a sophisticated technological base for new interface designs. Computer processor speeds support interactive applications that are increasingly innovative. And the increasing complexity of application software mandates a clear communication between user and computer (Fischer, 1989). But as Foley points out, "we are only beginning to understand what constitutes a good user interface and the management processes required to create such interfaces. ...there has been an insufficient software foundation upon which to build the interfaces" (Foley, 1986).

To reduce the high cost of implementing user interfaces, many research efforts explore development of improved tools for construction of user interfaces. The more advanced of these tools are referred to as User Interface Management Systems (UIMSs) (Pfaff, 1985). A UIMS implements some or all of the interface between the user and the application's action routines. Input to a UIMS typically includes screen designs, menu organizations, dialogue syntax, help files, and prompt messages. Interactive design tools are often provided so that the designer can specify these user interface elements graphically (Foley, 1986). Hill introduces the designation User Interface Development System (UIDS) to more accurately reflect the emphasis of current integrated environments in which to design, implement and test user interface software (Hill, 1986).

Graphical interfaces can ease the process of learning, using, and understanding applications, yet many applications do without a graphical interface because it is too difficult to construct. Even with the aid of a sophisticated graphics package, interfaces are

typically closely tied to their applications and therefore difficult to modify or reuse in different applications (Barth, 1986). Most user interfaces are implemented using traditional programming languages. Object-oriented programming provides a different, more powerful programming paradigm that enhances programmer productivity and encourages reuse of existing software modules (Foley, 1986).

A direct-manipulation interface presents its user with a set of visual representations on a graphical display for the internal objects and a repertoire of generic manipulations that can be performed on any of them. The user has no command language to remember beyond the standard set of manipulations and a continuous reminder on the display of the available objects and their states. Direct manipulation represents a powerful model for designing user interfaces (Jacob, 1986).

Traditional user interfaces are highly moded, that is the same input operations map to several different meanings. A moded interface requires that the user remember (or the system remind him) which mode the system is in at any given time and which different commands or syntax rules apply. Modeless systems do not require this (Jacob 1986). Direct-manipulation user interfaces appear to be modeless. Available objects are visible on the screen and the user can apply any of a standard set of commands to any object. The system remains in the same "universal" or "top-level" mode.

B. USER INTERFACE DEVELOPMENT SYSTEMS

The following sections discuss several User Interface Development Systems that have been implemented. Each demonstrates some of the principles that are important to user interface development. Other commonly known graphical user interface systems in use today include DECwindows, Motif, NewWave, NeXTStep, Open Look, Presentation Manager, SunViews, ViewPoint, and X Window System. Most of these provide tools and/or a development environment for designing and implementing graphical user interfaces.

1. Conman

Conman is a graphical Data-Flow Manager based on the UNIX principle of connecting simple tools that each do one thing well. After individual tools are started separately, the data-flow manager connects the output of one to the input of another. The

Conman environment facilitates creation of a user interface that translates user actions into higher level commands. It suggests an end to monolithic integrated applications in favor of dis-integrated functional fragments with protocols for communication of objects between fragments (Haeberli, 1987).

Conman runs under the Silicon Graphics Inc. IRIS window manager and presents a graphical representation of the input and output terminals of each active process as the user points to its window on the screen. The user completes a connection by selecting an output terminal, then selecting an input terminal using the mouse. After the connection is made, a displayed line connecting the output terminal to the input terminal indicates the connection is made. Connections may be broken later if required.

2. MIKE

MIKE is Menu Interaction Kontrol Environment, a UIMS developed by Brigham Young University Interactive Software Systems Laboratory. MIKE is implemented on a DEC/Vaxstation II and DEC VT240 graphics terminal and has a complete graphical layout facility within the interface editor for drawing the viewport and icon definitions. MIKE has the capability to generate a working prototype of the user interface with appropriate entry points for application specific procedures. The interface in development can be prototyped before application implementations are complete.

The development of MIKE was guided by five major goals applicable to UIMSs:

- A UIMS should be based on a simple conceptual model for designing user interfaces that is readily understood by both programmers and nonprogrammers alike.
- It should be possible to generate a working prototype of the user interface immediately on the basis of the basic definition alone.
- It should be possible to refine and enhance the default-generated interface continuously, using tools that are appropriate for the nonprogramming professionals who become involved in the design process.
- Where possible, all device dependencies, including those of a particular interactive style, should be isolated in the UIMS rather than in the dialogue description or the application.
- User interfaces generated by the system should be extensible in the sense that new commands and capabilities can be added without modifying the application code. (Olsen, 1986)

3. Sassafras

Sassafras is a UIDS consisting of five modules: an icon builder and library, an interaction module builder and library, a dialogue specification, an application interface specification, and an interface assembler. The icon builder and library construct and manage the symbols, icons and elements of user interface diagrams. The interaction module builder and library implements an interaction technique. The dialogue specification details the syntax of the language to be used between the user and the interface. The application interface specification describes the semantics of the system as a set of application routines implemented in some traditional programming language. The interface assembler compiles the pieces of the user interface, linking the dialogue specification with appropriate interaction techniques, icons, and application routines. Sassafras example applications include a simple paint program and a computer aided room layout program (Hill, 1986).

Sassafras supports an iterative design approach to user interface development. Using this approach, a designer first roughs out a user interface design and develops a prototype. On the basis of user performance using the prototype, the designer modifies the interface and implements a new prototype. The design-prototype-evaluate cycle repeats until the user interface is better than some standard specified by the designer. Hill points out that despite recommendations for this approach, it is rarely used because of its high cost. Time and money are usually exhausted during the first cycle (Hill, 1986).

4. GROW

GROW, the GRaphical Object Workbench, supports the development of graphical user interfaces that are highly interactive (including direct manipulation and animation). GROW simplifies the process of creating icons, linking the interface and application, and adding interactivity and animation. Interfaces can be modified and reused in other applications. Three techniques form the basis of GROW: object-based graphics with taxonomic inheritance, inter-object relationships such as composition and graphical dependency, and separation of the interface and application. Object-oriented programming and separation of the interface and application facilitate specializing and reusing interfaces.

Messages defined by GROW and the application provide interaction between the interface and the application program (Barth, 1986).

GROW is written in Interlisp-D, uses the object-oriented language Strobe and runs on Xerox 1100 series workstations. Applications using a GROW graphical interface include a data-flow program simulator, a Petri net editor and simulator, a program configuration editor, a constructive geometry editor, and an analog computer simulator.

5. Layered User Interface

The Layered User Interface (LUI) is a framework for generating consistent graphical interfaces composed of buttons, menus, sliders, and dialog windows. LUI is part of an image processing environment used at the Northrop Research and Technology Center. It is a series of programs coupled with a design methodology, rather than a single program. LUI allows incremental addition of user interfaces to graphics tools (Wilson, 1987).

An LUI "button" is a process with a small window attached. Pressing the button (with the mouse) causes a specific program to execute. Similar to a simple button, a "menu button" presents a menu of programs rather than a single choice. A "slider" is a process that translates the location of a slider bar into a UNIX command with a corresponding input value. LUI provides the "glue" for connecting tools together in ways that create new, more complex tools.

C. PREVIOUS NPS SIMULATOR INTERFACES

User interfaces developed to date for NPS 3D applications are similar in several ways, including the use of a mouse to make selections from menus and the use of the SGI button and dial box to control object selection and orientation. The menu facility provides pull-down, roll-off style menus supplied as a standard feature of the SGI operating system. The button and dial box controls are separate hardware devices that sit beside the display monitor and are linked to the application using operating system calls. These devices are optional equipment that must be purchased separately. Users that don't have this equipment are forced to either obtain the hardware or abandon those applications that require it.

1. Menu Systems

Using the standard menus on the Silicon Graphics hardware has several advantages, as well as a few disadvantages. The biggest advantage is ease of implementation: designing and implementing a menu system is a simple exercise with most of the effort spent designing the layout of the menu tree. Another advantage of the standard menu system is the ease of modification: adding a selection involves only inserting the selection into the appropriate location in the menu tree and specifying the associated action, while an item to be deleted is simply removed.

The major disadvantage of the SGI supplied menu system exists because a displayed menu receives the CPU as a dedicated resource, suspending all other operations. Thus applications suspend processing when a choice is being made from a menu. This is inconvenient at best, and unacceptable for real-time applications.

A second disadvantage is inherent to all pull-down menu systems: applications that require a large number of menu choices force the designer to use "roll-off" or "walking" menus. These menus incorporate a hierarchical design using sub-menus that are opened when a general menu item is selected. Each sub-menu opens to the right or left of its parent menu, and more than one sub-menu is possible. The problem occurs because the menu opens where the mouse is located at the time of the call (right mouse actuation), except when the mouse is located close to the borders of a window. At the edge of a window, the operating system will move the mouse away from the border far enough for the menu to be displayed. If there isn't enough room for the roll-off menus to be normally displayed, they will stack on top of one another rather than alongside the parent menu. The inconsistency of menu location and layout distracts the user's attention away from the application and violates basic user-interface guidelines.

2. Button and Dial Boxes

Button and dial boxes have other problems in addition to the question of availability. Poor feedback from the dials and difficulty setting a precise value because of the nature of their design commonly frustrate users. The mechanical rheostat device routinely produces inconsistent values. The button box offers immediate feedback by way of an LED display on each button, but as a mechanical device it is also subject to failure.

3. Colors

Another important feature of the interfaces for the NPS simulators is the use of color. In all of the simulators the colors are hard-coded, that is they can not be changed interactively by the user. This includes colors for the information and control panels, the terrain and objects (airplanes, ships, etc.), and the menus. In most cases these colors would never need to be changed. However the user interface should include the ability to customize the colors of a minimal set of things, including the simulator's controls and the information screens that define the look and feel of the application.

D. FOCUS -- NPS PANEL DESIGNER AND TOOLBOX

High quality interface software is costly in time and money, and it is essential for effective system performance. Our work concentrates on the graphical user interface and presents a study of information display, interface human factors, and underlying implementation efficiency considerations so as to enhance real-time simulation systems with minimal degradation in performance. Our research culminated in the implementation of the NPS Panel Designer and ToolBox (NPSPD), an automated development environment that enables design, implementation, modification, and testing of customized graphical user interfaces. NPSPD includes automatic generation of compilable source code which can stand alone or be integrated quickly into a developer's application. NPSPD was developed using Silicon Graphics Inc. IRIS 4D/70GT and 4D/GTX workstations, relatively low-cost systems which are commercially available. Methodology used and techniques developed provide a foundation applicable to any hardware capable of a windowing environment and graphics display.

E. THESIS ORGANIZATION

Chapter II presents human factors principles and guidelines indicated for successful user interface design. Chapter III presents the abstract functional model comprising NPSPD, including a description of its components, capabilities and terminology. Chapter IV presents the NPSPD design tools, including detailed descriptions of how they can be most effectively utilized. Chapter V describes the NPSPD ToolBox library of actuators and functions, including examples of programming level usage. Chapter VI describes the

NPSPD File Manager and the intermediate file with its format, flexibilities and modification techniques. Chapter VII describes the NPSPD Code Generation feature with suggested entry points and instructions for linking panel code to target applications. Chapter VIII presents the development of a complete graphical user interface for an existing application. Chapter IX presents capabilities and limitations of NPSPD with suggestions for further research and possible system enhancement. Chapter X presents conclusions. Appendix A presents the NPSPD User's Guide and Appendix B presents the NPSPD Reference Manual. Appendix C lists reserved words for the NPSPD intermediate file parser. Appendix D presents sample generated code. And Appendix E presents a sample intermediate file.

NPS Panel Designer and ToolBox was in all respects a team effort by both authors, each contributing to the details of the others work. Both authors wrote and reviewed all aspects of this work. Lieutenant King developed the panel, actuator, and color editing modules and designed the intermediate file format including the lexical analyzer and parser central to saving and recalling panel design files. His major focus was the development of the Panel Designer interface control, information prompts and the help system. His writing focused in Chapters I, IV, VI, VII, VIII, IX, and X. Lieutenant Commander Prevatt designed the data structures and functions used in the Panel Designer and ToolBox, designed the attributes of panels and actuators, implemented the processing and drawing routines, and developed the source code generation module. He provided the overall program design including the format for the generated source code and the means to link it to users' applications. His writing focused in Chapters I, II, III, V, IX, and X.

II. HUMAN FACTORS AND USER INTERFACE DESIGN

> "A part of the purpose of the user interface is to transform data into information and present it in a fashion that makes it easily absorbed and used" (Smith, 1963, p.297).

The user interface is the method used by an application program to interact with the operator (Goodwin, 1989, p.viii). Its main function is communication. The user interface must convey to the user all the instructions on the program's use, must allow the user to control the program as naturally as possible, and must provide the program's results to the user. If the interface gives unclear instructions, or complicates the control of the task, then the communication, the interface, and ultimately the software are unsuccessful (Brown and Cunningham, 1989, p.6). Users may compensate for a poor design with extra effort, and probably no single user interface design flaw, in itself, will cause system failure. But there is a limit to how well users can adapt to a poorly designed interface (Smith and Mosier, 1986, p.3).

Software is not the only significant factor influencing user performance. Other aspects of user interface design are also important, including workstation design, physical display characteristics, keyboard layout, environmental factors such as illumination and noise, written documentation, and training (Smith and Mosier, 1986, p.2). It is useful to distinguish between the physical and conceptual aspects of the interface. The hardware (keyboard, display screen, mouse, etc.) affect the way in which the system behind the interface may be used, but they do not affect its conceptual power. Conceptual limitations arise because the machine representation of the world is minimal and rigid, and usually only just sufficient to achieve the task in hand (Thimbleby, 1985, p.168).

We will focus on design features of the user interface that are implemented via software. "The 'design' in the program establishes the contents of processed data available to the operator and the visual relationships among the data. It can also establish the sequence of actions which the operator must use and the feedback to the operator concerning those actions" (Parsons, 1970, p.169). The developer must distinguish between

what the computer will do and what the human will and can do. He makes assumptions about what the user will want to do, remembering that each user is an individual, with his or her own talents, goals, knowledge and preferences (Fischer, 1989).

A. DESIGN PRINCIPLES

1. Guidelines and References

As formal human-computer interface theories are fully developed, researchers propose applicable methods and principles of interface design (Fischer, 1989). Ramsey and Atwood completed a comprehensive survey of user-computer interaction literature as early as 1979. MIL-STD-1472D, revised in 1989, provides minimal guidance for the interface developer in a relatively small section, "User-Computer Interface". Several organizations, including those listed below, have developed helpful in-house guidelines for user interface design.

- *Guidelines for Designing User Interface Software.* by MITRE Corporation for Electronic Systems Division, U. S. Air Force.
- *Spacelab Experiment Computer Application Software (ECAS) Display Design and Command Usage Guidelines.* NASA (National Aeronautics and Space Administration).
- *Human Factors in Office Automation.* Life Office Management Association.
- *Human Factors Engineering Standards for Information Processing Systems.* Lockheed Missiles and Space Company.
- *Design guidelines for user transactions with battlefield automated systems: Prototype for a handbook.* US Army Research Institute.

2. The Development Process

User interface guidelines vary depending on the application and the target users' skill levels. Even the most careful design will require testing with actual users in order to confirm the value of good features and discover what bad features may have been overlooked (Smith and Mosier, 1986, p.10). "Neither the Designer nor the User have a clear idea of what is required until they have a working system" (Thimbleby, 1985, p.169). Testing is so essential for ensuring good design that some experts advocate early creation of an operational prototype to evaluate interface design concepts interactively with users,

with iterative design changes to discover what works best (Gould, 1983). But prototyping is no substitute for careful design. Prototyping will allow rapid change in a proposed interface, but unless the initial design is reasonably good, prototyping may not produce a usable final design (Smith and Mosier, 1986, p.10).

We have seen the design-prototype-test-redesign principle practiced in current NPS Computer Science research as thesis students use NPSPD to work out an initial user interface design for their system, then evaluate it in the context of the application, and subsequently return to NPSPD to redesign and improve the interface.

3. Consistency, Flexibility and the User Model

The interface should be consistent throughout the program supporting a single model of the problem and its solution. The user needs a concise, memorable and accurate maxim which conveniently expresses the rules of system interaction. Together with training in explicit reasoning about interaction, the maxim provides a sure foundation on which to build the user model. The designer then uses the same maxim (plus equivalent reasoning processes) to constrain system behavior to be compatible with it (Thimbleby, 1985, p.171). One successful interface technique provides that 'what you see is what you get'. This phrase specifies certain properties of a user interface which, with a little explanation, may be used for the user to develop hypotheses about system behavior (Thimbleby, 1985, p.175).

Consistency includes two aspects--consistency in the mental model a user has of an application, and consistency in the way the user controls the application (Brown and Cunningham, 1989, p.9). Consistent sequences of actions should be required in similar situations, identical terminology should be used in prompts, menus, and help screens, and consistent commands should be employed throughout. Exceptions should be comprehensible and limited in number (Shneiderman, 1987, p.61).

The interface must be flexible to work with a wide range of users without sacrificing consistency (Brown and Cunningham, 1989, p.7). A flexible interface provides multiple ways to accomplish the task and allows the user to choose aspects of the interaction style which he or she prefers (e.g. menu versus function key selection). To some

extent, the user should be able to establish the interaction method and to customize the interface.

4. Informative Feedback and Help

The interface must keep the user aware of what is going on in the task (Brown and Cunningham, 1989, p.8). For each action there should be appropriate system feedback. For frequent and minor actions the response can be very modest, and for infrequent and major actions the response should be more substantial. Visual presentation of the objects of interest provides a convenient environment for explicitly showing changes (Shneiderman, 1987, p.61). The computer should provide suitable cues to allow the user to select the next action, e.g. a confirmation for important processing such as deleting an object or file. The system must provide timely confirmation that desired actions are in progress or completed (Fischer, 1989).

The user interface should include access to help. A well-constructed interface can almost totally eliminate the need for an external manual. A multi-faceted help facility appears to be best at providing incrementally more assistance as may be needed by expert to novice users. Quality manuals, online help, and tutorials have a profound effect on users' success and their impressions of most interactive systems (Shneiderman, 1987, p.382).

In an interactive computing situation, immediate feedback by the system is important in establishing the user's confidence and satisfaction with the system. A message that indicates that the system is still working on the problem or a signal that appears while the system is processing the user's input provides the user with the necessary assurance that everything is all right. Predictability of computer response is related to system response time. Timely response can be critical in maintaining user orientation to the task. Some experts argue that consistency of system response time may be more important in preserving user orientation than the absolute value of the delay, even suggesting that designers should delay fast responses deliberately in order to make them more consistent with occasional slow responses (Engel, 1975, p.13).

5. Dialogues that Yield Closure

Sequences of actions should be organized into sequences with a beginning, middle, and end. The interface should provide informative feedback at the end of a group of actions (closure). This feedback gives the operator the sense of completion or accomplishment and suggests that he prepare for the next sequence of actions (Shneiderman, 1987, p.61).

6. Shortcuts for Expert Users

Humans begin as novices and progress with experience to higher levels of user classification (Fischer, 1989). As frequency of use increases, so does the desire to reduce the number of interactions and increase the pace of interaction. Abbreviations, special keys, hidden commands, and macro facilities assist frequent and knowledgeable users. Shorter response times and faster display rates improve productivity and the feeling of system responsiveness (Shneiderman, 1987, p.61). However, consistency of control is more important than shortcuts. The user depends on consistent interface design to set practical limits on what must be learned and remembered about the system (Smith and Mosier, 1986, 213).

Two forms of human memory affect user interface design. Recognition memory connects command choices to displayed options, as in a menu. Recall memory enables selection of an invisible command choice or option for the desired action when the user already knows the commands (Brown and Cunningham, 1989, p.22). Experienced users are more able to rely on recall memory as is needed for command systems. Recognition memory is preferable for novice users.

7. Internal Locus of Control

Operators desire the sense that they are in charge of the system and that the system responds to their actions. Surprising system actions, tedious sequences of data entries, difficulty in obtaining desired information and inability to produce the desired action all build anxiety and dissatisfaction. Users should be the initiators of actions rather than the responders (Shneiderman, 1987, p.62). And the interface should keep the computer from coming between the user and the work (Brown and Cunningham, 1989, p.7).

8. Reversal of Actions

As much as possible, actions should be reversible. This relieves user anxiety since errors are easily correctable, and it encourages exploration of unfamiliar capabilities and options (Shneiderman, 1987, p.62). A user may not know what he wants to do as he begins his work. The interface can supply the freedom to experiment and then "UNDO" a mistake.

9. Simple Error Handling

An effective interface design ensures the user cannot make a serious error. If an error is made, the system detects it and offers simple, comprehensible mechanisms for handling the error. The user should not have to retype the entire command or entry, but only repair the faulty part. Erroneous commands should leave the system state unchanged or produce instructions about restoring the correct system state (Shneiderman, 1987, p.61). Error messages should be clear, concise and without accusation in tone. Help when needed should be available.

"The program must not crash" (Brown and Cunningham, 1989, p.8).

10. Minimal Short Term Memory Load

Humans can rapidly recognize approximately seven plus or minus two "chunks" of information at a time and hold them in short-term memory for fifteen to thirty seconds. The size of the chunk depends on familiarity with the information. This has been found to be the case for a number of different scales, for example, color, size, brightness, loudness and so on (Miller, 1956, pp.81-97).

The limitation of human information processing and recall in short-term memory requires that interfaces be simple. Reduced window motion and sufficient training time for codes, mnemonics and sequences of actions improve user performance. Online access to command syntax forms, abbreviations, codes, and other information should be provided (Shneiderman, 1987, p.62). Even an experienced user will spend time away from the application. The interface can assist while he returns to his former level of expertise.

Another difficulty stems from confusion between functionality and ease of use. A system becomes easier to use for a designer as more capability is added. But the designer has a higher threshold for complexity than the user, especially when the user is learning.

The system becomes harder to use for the user the more that needs to be learned and the more that can be done accidentally (Thimbleby, 1985, p.169).

B. ASPECTS OF THE USER INTERFACE

1. Data Entry

Data entry refers to user actions involving input of data to a computer, and computer responses to such inputs--selecting an object, designating a position, text entry, actuator control, etc. On-line data entry provides the opportunity for immediate system validation of user inputs, with timely feedback so that the user can correct errors. Another important design concept is flexibility--the interface should adapt to the users needs. Pacing of input should be dictated by the user rather than the system. General objectives for data entry design are consistency of data entry transactions, minimizing input actions and memory load on the user, ensuring compatibility of data entry with data display, and providing flexible user control of data entry. Data entry guidelines as adapted from Smith and Mosier follow (Smith and Mosier, 1986, pp.11-90).

- Data entry in context--depth values for submarines, altitudes for airplanes.
- Data entered only once--system should reference original input values.
- Entry via primary display--entered data should appear on primary display.
- Feedback during data entry--display feedback for all user actions.
- Fast response--system feedback should not exceed 0.2 sec (Engel and Granda, 1975)
- Single method for entering data--one consistently available method of data entry.
- Defined display areas for data entry--clear visual definition of entry fields.
- Consistent method for data change--display previous data allowing type-over/insert.
- Explicit Enter action--require user to explicitly accept entry for processing.
- Explicit Cancel action--allow user to explicitly cancel entry before processing.
- Feedback for completion of data entry--system acknowledgment of success or error.
- Distinctive cursor--movable cursor with distinctive visual features.
- Non-obscuring cursor--cursor should not obscure other data on display.
- Precise pointing--cursor includes precise designation for accurate position selection.
- Explicit activation--explicit action separate from cursor positioning to accept entry.

- Fast acknowledgment of entry--feedback should not exceed 0.2 sec.
- Large pointing area for option selection--allow area for pointing and selection to be as large as consistently possible.
- Pointing--when graphic data entry involves pointing, design the user interface so that actions for display control and sequence control are also accomplished by pointing, i.e. single method of entry, single entry device.

2. Data Display

Data display refers to computer output of data to a user, and assimilation of information from such outputs. Display is particularly critical in control tasks involved in simulation. Avoid overfilling the screen. Twenty-five percent full is considered to be the maximum above which the background "noise" reduces the ability of the user to locate and recognize information (Reid, 1985, p.114). Use the upper right hand quadrant of the screen for exceptional information. Danchak (1977) reports that users are more sensitive to changes in the upper right hand quadrant than either of the left hand quadrants. Users are least sensitive to changes in the lower right hand quadrant. Objectives of data display include consistency of data display, efficiency of information assimilation by the user, minimal memory load on user, compatibility of data display with data entry, and flexibility for user control of data display. Data display guidelines as adapted from Smith and Mosier follow (Smith and Mosier, 1986, pp.91-209).

- Display all and only necessary data--do not overload displays with extraneous data.
- Consistent display format--display data consistently from one screen to the next.
- User control of data display--allow users to control the amount, format and complexity of displayed data as necessary to meet task requirements.
- Standard symbols--establish standard meanings for graphic symbols and use them consistently within a system and among systems with the same users.
- Provide overview position of visible section--when user pans over extended display, provide some graphic indicator of position of visible section within overall area.
- Aid distance judgment -- provide computer aids to distance judgment within graphic display when accurate distance perception is important.
- Consistent format-- adopt a consistent organization for the location of various display features from one display to another.
- Distinctive display elements -- make different elements of a display distinctive from one another.

- Spacing to structure display -- use blank space to structure a display. Do not over-crowd data.
- Page crowded displays--when a display contains too much data, separate into selectable pages.
- Related data on same page -- keep functionally related data on the same display page.
- Conservative use of color -- employ color coding conservatively, using relatively few colors and only to designate critical categories of displayed data. Limit the number of colors to seven in the entire sequence of screens and choose color combinations carefully.
- Tonal coding--where users must make relative judgments for different colored areas of a display, consider employing tonal codes (different shades of one color) rather than spectral codes (different colors) (Phillips, 1982).
- Ordered color coding--where different areas of a map are coded by texture patterns or tonal variation, order the assigned code values so that the darkest and lightest shades correspond to the extreme values of the coded variable. Darkest blue for deepest ocean depth and lightest blue for shallowest.
- Highlighting--if one area of map is of particular interest, highlight that area.
- Color coding to support task--color tailored to task speeds recognition.
- Color coding under user control--allow user to select and set color coding.
- Redundant color coding -- make color coding redundant with some other display feature such as symbology and do not code only by color.
- Unique assignment of color codes -- when color coding is used, ensure that each color represents only one category of displayed data consistently.

3. Sequence Control

Sequence control refers to user actions and computer logic that initiate, interrupt or terminate transactions (Smith and Mosier, 1986, p.211). One of the critical determinants of user satisfaction and acceptance of a computer system is the extent to which the user feels in control of an interactive session. If users cannot control the direction and pace of the interaction sequence, they are likely to feel frustrated, intimidated or threatened by the computer system. Their productivity may suffer, or they may avoid using the system at all (Brown, 1983, p.4-1).

A fundamental decision in user interface design is selection of the dialogue types(s) that will be used to implement sequence control. And an important aspect of

dialogue choice is that different types of dialogue imply differences in system response time for effective operation. Menu selection, function key selection, and graphic interaction all require fast system response (Smith and Mosier, 1986, p.212). Menus have been recommended for occasional and novice users as they reduce the amount of information the user needs to remember. They also serve the useful function of limiting, to a well defined set, the responses the user can make. Their disadvantage is that, particularly for the more experienced user, they may be ungainly. As well as being frustrating this can lead to problems navigating through complex systems (Reid, 1985, p.111).

Consistency of control is more important than shortcuts. The user depends on consistent interface design to set practical limits on what must be learned and remembered about the computer tools. Objectives of sequence control include consistency of control actions, minimizing control actions by the user, minimizing memory load on the user, ensuring compatibility with task requirements, and providing flexibility of control. Sequence control guidelines as adapted from Smith and Mosier follow (Smith and Mosier, 1986, pp.211-290).

- Minimize user actions -- ensure that control actions are simple, particularly for real-time tasks requiring fast user response.
- Match control to user skill -- simple step by step control for novice users and short-cut or complex interaction for experienced users.
- Compatibility with user expectations -- ensure that the results of any control entry are compatible with user expectations and do not confuse the user.
- Supplementary verbal labels for icons -- if icons are used to represent control actions in menus, display a verbal label with each icon to help convey its intended meaning.
- Direct manipulation -- provide a capability for direct manipulation of objects as a means of sequence control.
- General list of control options -- provide a general list of basic control options that are always available to user to serve as a consistent foundation for system control.
- Indicate appropriate control options -- make available a list of options that are valid.
- Prompt for control entry -- guide control entries in sequence as needed by user.
- Display most likely options first -- except as dictated by consistency of control.
- Appropriate response to all entries -- design the interface to deal appropriately with all possible control entries, correct and incorrect.

- Warn of potential data loss or irrevocable change -- prompt user to explicitly confirm actions that result in loss or change to data.
- Provide an UNDO function -- allow easy reversal of actions.

4. User Guidance

User guidance refers to error messages, alarms, prompts and labels, as well as to more formal instructional material provided to guide the user's interaction with the computer. Goals are to permit efficient system use (quick and accurate use of full system capabilities), with minimal memory load on the user and minimal time required to learn the system. User guidance should be regarded as a pervasive and integral part of interface design that contributes significantly to effective system operation. Good user guidance results in faster task performance, fewer errors and greater user satisfaction. Goals of user guidance include consistency of operational procedures, efficient use of full system capabilities, minimal memory load on user, minimal learning time, and flexibility in supporting different users (Smith and Mosier, 1986, p.291).

Much of the information commonly provided in paper documentation, such as user manuals, should also be available on line (Brown 1983, p.6-1). Further, on-line documentation offers a potential cost savings of 70 to 80 percent over more traditional paper documentation (Limanowski, 1983). User guidance guidelines as adapted from Smith and Mosier follow (Smith and Mosier, 1986, pp.291-336).

- Standard procedures -- design standard procedures for similar, logically related transactions.
- Explicit user actions -- require user to take explicit actions to initiate processing.
- Affirmative statements -- adopt affirmative rather than negative wording for user guidance messages.
- Active voice -- rather than passive voice in user guidance messages.
- Temporal sequence -- preserve temporal sequence of steps in wording of user guidance about that sequence.
- Consistent grammatical structure
- Flexible user guidance -- provide means for experienced user to by-pass standard or lengthy guidance procedures.

- Informative error messages -- when the system detects an error, display an error message to the user stating what is wrong and what corrective action can be taken.
- Brief error messages -- be clear, concise and informative. Extra words are not helpful.

5. Data Protection

Data protection attempts to ensure the security of computer-processed data from unauthorized access, from destructive user actions, and from computer failure. Goals of data protection include effective data security, minimal entry of wrong data, minimal loss of needed data, and minimal interference with information handling tasks. If data loss from machine failure and data loss from faulty system operation are minimized through careful design, then the most serious threat to data protection is the system user (Smith and Mosier, 1986, p.371).

III. NPSPD FUNCTIONAL MODEL

NPS Panel Designer provides a powerful environment for rapid design, development and testing of graphical user interfaces. NPSPD customization tools enable the developer to tailor an interface to the exact needs and specifications of an application. This chapter presents the abstract functional model comprising NPSPD including a description of its components, capabilities and terminology. Chapters IV and V, close companions to this chapter, describe the NPSPD Design Tools and the NPSPD ToolBox of actuators and functions. The NPSPD User's Guide, Appendix A, and the NPSPD Reference Manual, Appendix B, provide complete details of the structures, functions and usage procedures.

Although an object oriented language such as C++ was not available during the design and implementation of the NPS Panel Designer and ToolBox, an object oriented approach was used. Distinct abstract data types for the basic actuator and all detailed actuators are defined. Object related functions provide access to panel and actuator attributes, details and values.

A. Model of the User Interface

The Panel ToolBox provides customized windows called panels and pre-designed, customizable, mouse-sensitive controls called actuators. A graphical user interface implemented using NPSPD consists of one or more panels each having zero or more actuators positioned and functionally connected according to application needs. NPSPD itself was developed and finalized using earlier versions of NPSPD. Figure 3.1 presents an example interface developed using NPSPD.

The ToolBox enables the user to control the interface via the mouse or optional user-defined keyboard equivalent keys. As the user presses the left-mouse button with the mouse-cursor inside the boundary of an actuator, the ToolBox records the actuator as selected and active, and it records the host panel associated with that actuator as selected and active. A panel can be selected even without the direct selection of one of its actuators

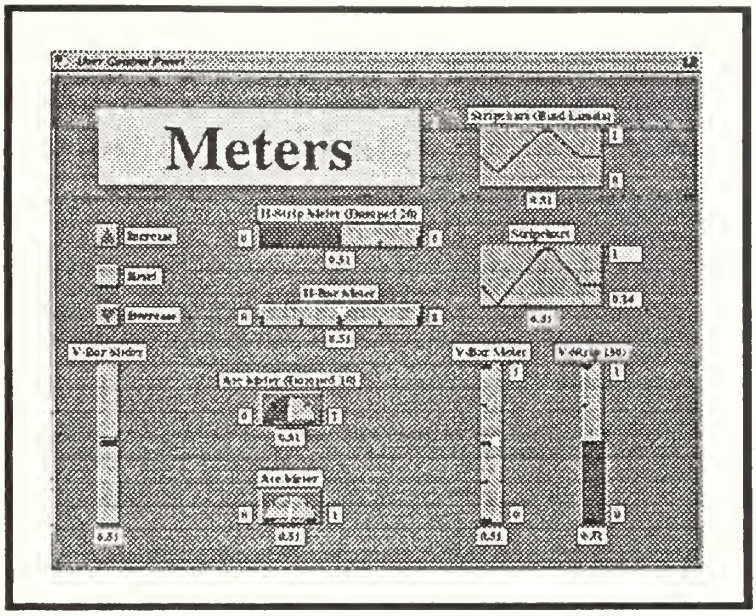

Figure 3.1 Sample Interface Developed using NPSPD

by pressing the left-mouse button with the mouse-cursor on the panel but outside of all of the actuators. A panel and actuator remain selected until the left-mouse button is released.

Positioning the mouse-cursor while the left-mouse button is pressed controls the value of an actuator depending on the nature and function of that actuator. The actuator's displayed appearance reflects its state and value at all times. While the value or state is changing, the ToolBox redraws the actuator. Once the panel or actuator is de-selected, the ToolBox updates the display one final time and not again until the user initiates some other control action.

NPSPD provides key-equivalents for activation of actuators. A key-equivalent is an optional, user-defined key that is associated with an actuator. Pressing the defined key causes the same processing as activation with the left-mouse button. Key-equivalents apply primarily to buttons.

Any time a panel or an actuator is selected, the ToolBox selectively executes several optional, user defined functions. Pointers provided within the data structures of each panel and each actuator reference a processing function, a new value function, a left-mouse button down function, an active function, and a left-mouse button up function. During each interface processing cycle for a selected actuator, *processfunc*[1] (if defined) performs

internal processing that must occur every cycle, *newvalfunc* computes the actuator state and value based on the X and Y coordinates of the mouse-cursor relative to the actuator, and *downfunc*, *activefunc* and *upfunc* connect user defined functionality to the actuator. The NPSPD Reference Manual provides details of these functions and their uses.

B. Development Process

Development of an effective interface requires a thorough consideration of the application to which it will be applied. Five basic phases make up the development process: preliminary design of the interface content and layout, development of the interface in the NPSPD environment, generation of compilable source code, modification of the interface and application source code to include appropriate communication links, and finally, compiling and linking the NPSPD interface code with the application code.

Once the application needs are defined, a careful layout sketch clarifies the user interface and speeds the development process using NPSPD. Most interfaces are laid out in screen relative units (pixels). For the SGI standard 19 inch display screen, 100 pixels span approximately one inch. Quarter inch ruled graph paper is well suited to preliminary graphical layout of the user interface panels.

After the initial analysis and design, NPSPD is executed and used to create the required panels and actuators for an interface. The panels and actuators are customized as to location, size, label, value display, colors, etc. At convenient times during the development, the NPSPD interface layout can be saved to an intermediate file for later recall and modification. We recommend saving the interface during development because this provides backup versions in case the computer malfunctions or some of the design modifications are deemed inappropriate and the developer decides to return to an earlier version of the design.

Once the design is sufficiently implemented within NPSPD to warrant testing, code generation produces compilable C-language source code in three files: *User_panel.h*, *User_panel.c* and *User_panel_fn.c*. The developer introduces the interface modules into

---

[1] In the text of this thesis, italicized text refers to procedures, variables or statements from NPS Panel Designer and ToolBox source code.

the application or the application code into the interface control module or both. Chapter 7 presents a detailed discussion of the code generation and application linking.

The developer compiles and links the user interface and application. He tests the user interface in the context of the application and feeds the results back into a redesign of the interface. NPSPD may be used repeatedly to refine and expand the interface design. Most initial NPSPD interface implementations can be produced in a matter of a few minutes. Then time may be devoted to the details of the application and the fine points of the user interface. Refinements and improvements are easily implemented.

C. NPS Panel Designer

The NPSPD environment, shown in Figure 3.2, consists of a Palette of actuators and one or more workspace panels. The opening NPSPD copyright panel remains displayed during the initialization sequence, approximately 3 seconds.

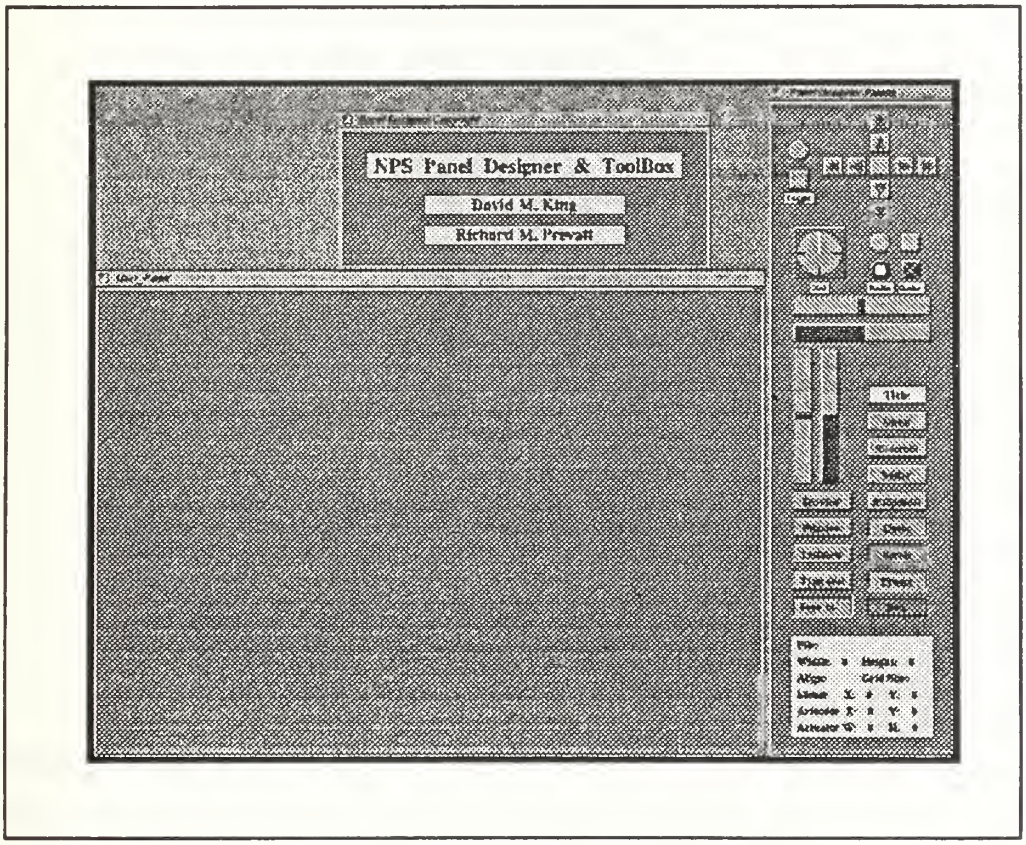

Figure 3.2 Opening Layout of PD

1. Palette and Actuators

The Palette, depicted in Figure 3.3, presents all of the actuators provided by the Panel ToolBox for development of user interfaces. The representations for the Buttons, Dials and Sliders are default versions of each of those actuators. All other actuators are made available via labeled selection buttons.

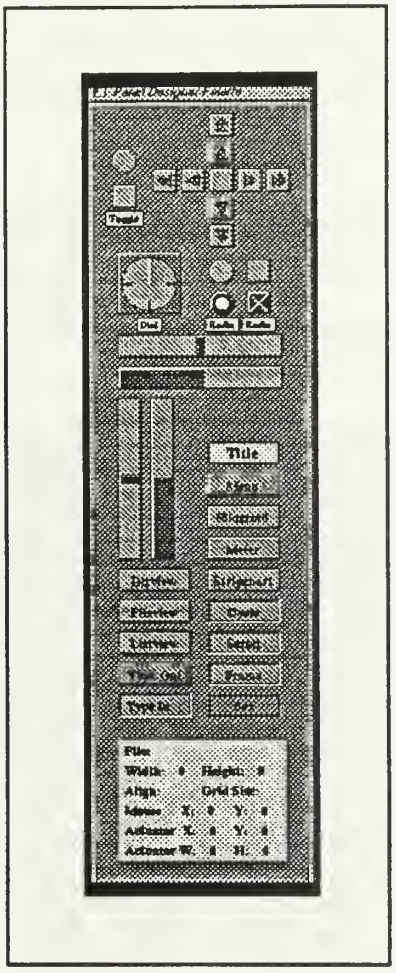

Figure 3.3 NPSPD Palette

In the lower portion of the Palette, the Workspace Status Display presents the name and size of the current workspace, the status of workspace auto-alignment and layout grid size, the location in panel coordinates of the mouse cursor, and the location and size of

the current actuator on the current workspace. These fields provide continual readouts of layout information useful to the developer.

ToolBox actuators include: momentary buttons, toggle buttons, horizontal and vertical sliders, dials, menus, file-views, list-views and directory-views, custom frames, outline boxes, type-in and type-out fields, meters and stripcharts. Table I presents a complete list of the basic types of ToolBox actuators. The NPSPD Reference Manual provides a detailed description of each actuator.

box	meter
button	scroll
cycle	slider
dial	slideroid
dirview	stripchart
fileview	title
frame	typein
listview	typeout
menu	

Table I ToolBox Actuators

A basic actuator abstract data-type provides the foundation for all of the diverse ToolBox actuators. Attributes are properties common to all actuators and are recorded in the actuator base structure. Attributes include location and size, value, minimum and maximum values allowed, label, value display format, etc. Each actuator adds unique details to the basic attributes. Details are recorded in a detail structure specific to each different actuator and allow for variation of appearance and function within types of actuators. As an example, the details associated with a Dial include the shape (CIRCLE or RECTANGLE), the number of major and minor tics on the Dial face, and the fine control factor. The NPSPD Reference Manual presents a complete description of actuator attributes.

2. Workspaces and Panels

Within NPSPD, a workspace is any one of the set of panels onto which the developer positions actuators. It is the blank slate on which the developer designs the user

interface. Other panels such as the Palette, Actuator Editor, Color Editor, Panel Editor, etc. are a part of NPSPD but are not available as workspaces.

When NPSPD is initiated, a single workspace panel is presented. Any number of additional workspace panels may be created and modified to participate in the interface under development. All workspaces may be cleared or deleted according to the developer's desires. Each workspace panel exactly represents the user interface panel generated by the code generator. Functionality must be included by the application developer.

D. Interaction with NPSPD

NPSPD supports three means of interaction control: direct manipulation using the mouse, feature selection using the keyboard and feature selection using pop-up menus. The mouse provides control of interface layout, actuator placement and actuator modification. Function keys and selected special keys of the keyboard provide the primary means for selection of design tools, editors and managers. Pop-up menus provide an alternate means of selection.

1. Mouse

The mouse consists of the on-screen cursor and the mouse control unit with its optical sensor, reference pad and three selection buttons. The mouse-cursor is displayed as an arrow in the Palette and as a cross inside all workspace panels. "Left-mouse", "middle-mouse" and "right-mouse" refer to the left, middle and right mouse buttons, respectively, in conjunction with the mouse-cursor position. The location of the mouse determines the current panel and current workspace.

a. Left-mouse

The left-mouse controls the operation of actuators (e.g., toggle buttons, slide sliders, or set dials). Left-mouse down activates an actuator and its associated host panel, or the panel only if the mouse-cursor is not on an actuator. Left-mouse up de-activates the actuator and/or the associated panel. The left-mouse functions both within NPSPD and within generated user interfaces.

b. Middle-mouse

The middle-mouse selects an actuator as current within an NPSPD workspace or the Palette. Pressing and releasing the middle-mouse selects an actuator. Pressing and holding the middle-mouse moves or re-sizes an actuator. The middle-mouse functions only within the NPSPD environment and NOT within generated user interfaces.

c. Right-mouse

The right-mouse controls menu selections. Pressing the right-mouse within any workspace pops up the NPSPD main menu of tools, editors and managers. Positioning and releasing the right-mouse while the desired choice is highlighted activates NPSPD processing associated with that menu choice. The right-mouse functions both within NPSPD and within generated user interfaces.

2. Keyboard

NPSPD provides direct access to all of its tools, editors and managers via function keys as described in Table II. Experienced developers speed the development process by

F1	On-line Help Manager
F2	Actuator Auto-alignment
F3	Layout Grid Display
F4	Layout Grid Size
F5	Create New Workspace
F6	Clear Current Workspace
F7	Delete Current Workspace
F8	Panel Editor
F9	Actuator Editor
F10	Color Editor
F11	Intermediate File Manager
F12	Source Code Generation Manager
Insert	Copy the current workspace actuator if any
Delete	Delete the current workspace actuator if any
Backspace	Delete the current workspace actuator if any
Ctrl	Fine control of actuator value
Esc	Exit NPS Panel Designer

Table II NPSPD Keyboard Functions

use of the function keys rather than the pop-up menu system. NPSPD includes both in keeping with the flexibility requirements of an effective user interface. The insert, delete and backspace keys are active to provide direct actuator copy and delete functions on a workspace. The control key (Ctrl) modifies the behavior of some actuators to yield a fine control operation. Escape provides direct exit from the Panel Designer.

3. Menu

NPSPD provides alternate access to design tools and features via pop-up menus. Table III presents the NPSPD menu selection hierarchy. Upon pressing the right-mouse button within any workspace, NPSPD presents the main menu. Sub-menus appear as the developer makes a roll-off selection.

Main Menu Selections:	Sub-menu Selections:
Layout Tools...	
	Auto Align On/Off
	Layout Grid On/Off
	Set Grid Size
Workspace Tools...	
	Create new Workspace
	Clear Current Workspace
	Delete Current Workspace
Panel Editor	
Actuator Editor	
Color Editor	
File Manager	
Code Generation	
Quit	

Table III NPSPD Menu Selections

4. Current Workspace and Actuator

NPSPD denotes the workspace on which the mouse-cursor is located as the current panel and the current workspace. Design tool and editor actions take effect in the current workspace. If the mouse-cursor is on the Palette or outside of all of the panels, there is no current panel or current workspace.

Each NPSPD panel may have one actuator selected and designated as the current actuator. Selection via the middle-mouse button displays a white highlight outline around the body of the actuator. NPSPD references the current actuator of the Palette when adding new actuators to a workspace using the middle-mouse button.

5. Workspace Tools

NPSPD provides three tools for managing the workspace environment. They are create a new workspace, clear an existing workspace and delete an existing workspace. Workspace tools are available directly using function keys as described in Table II or via the NPSPD pop-up menu using the right-mouse.

6. Customization and Layout Tools

NPSPD provides customization tools to support detailed design of panels and actuators. These tools include the Panel Editor, Actuator Editor and Color Editor. Chapter 4 describes each tool. NPSPD also provides selectable auto-alignment of the actuators on a workspace panel. When auto-alignment is on, NPSPD moves all actuator origins (lower left corner) to the nearest layout grid intersection. New actuators also align to the grid. Grid size is selectable from a sub-menu as 5, 10, 25, 50, 75 or 100 panel units. The layout grid may be displayed independently of the auto-alignment feature.

E. Addition, Deletion, Modification of Panels and Actuators

A workspace may be positioned anywhere on the screen using the left-mouse on the panel's window border. The panel is resized by pressing the left-mouse while the cursor is on any one of the corner resize handles, then dragging the window outline to a desired shape. If the border has been de-selected for a particular panel, that panel may not be moved or resized. The Panel Editor discussed in Chapter IV enables modification of all of the panel attributes. NPSPD workspace tools enable addition and deletion of workspace panels providing a confirmation prompt before the action is finalized.

Actuators may be added to a workspace in two ways, from the Palette using the middle-mouse or from the current workspace using the NPSPD copy tool. The left-mouse button is used to select an actuator icon on the Palette as current. The mouse-cursor is positioned on the workspace at the location for the origin of the new actuator and the

middle-mouse button is pressed and released. NPSPD creates and positions a new actuator at the specified location. The origin of each actuator is its lower left corner. The alternate way to add actuators to a workspace panel is to select an actuator on the workspace as current using the left-mouse button. Pressing the Insert key or selecting the copy option from the Layout Tools sub-menu causes NPSPD to create an exact duplicate of the current actuator. The new actuator is positioned above and to the right of the original one.

Actuators may be moved and resized on a workspace by placing the mouse-cursor on the actuator and holding the left-mouse button down. Figure 3.4 maps the selection areas associated with each actuator body to the resulting NPSPD modification.

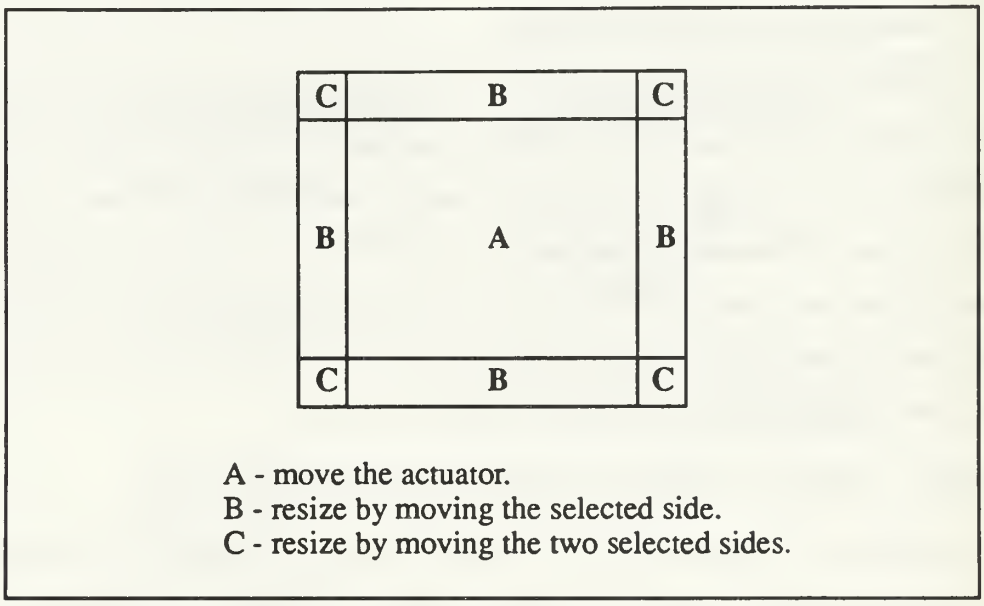

Figure 3.4 NPSPD Actuator Move/Resize Areas

F. Intermediate File

The File Manager feature of the NPSPD, discussed in Chapter VI, enables the user to save and recall workspace designs. NPSPD writes all of the pertinent information for a workspace to an ASCII file called the intermediate file. This highly structured file enables the user to store and recall uncompleted work, combine two or more separate designs, and modify designs manually (outside of the NPSPD environment) by using any text-based editor. Appendix E presents a sample intermediate file.

G. Source Code Generation and Application Linking

One of the most powerful features of the NPSPD is its ability to generate source code that corresponds to an interface design. Using the Code Manager as described in Chapter VII, the developer generates source code for the current workspace or all workspaces. The code may then be modified to communicate with the application using clearly defined entry points. The modified code is compiled and linked with the application, providing a custom interface.

There are two methods of integrating an interface designed with the NPSPD into an application. The first method uses the framework of the code generated by the NPSPD and integrates the target application's control features using the NPSPD provided entry points. We recommend this technique for users that are designing an application from the beginning.

The second method involves integrating an interface designed with the NPSPD into an existing application by discarding the bulk of the NPSPD code generated for the interface and using only those functions necessary to initialize, control and draw it. This technique integrates a graphical user interface into applications that either don't have one, or have one that is considered inadequate. Chapter VIII presents a complete NPSPD application. Appendix D presents sample code generated by the NPSPD.

H. Compilation

Figure 3.5 presents an example of the instructions required to compile the interface source code produced by NPSPD. The Panel ToolBox library, *npspanel.a*, must be available to the developer via an appropriate directory path as shown.

```
cc -o user_name User_Panel.c User_Panel_fn.c /nps_path/lib/npspanel.a
                -I/nps_path/include -O2 -align16 -G 0 -lc_s -lgl_s -lfm -lm
```

/nps_path must be defined as the proper path to the NPS Panel ToolBox.

/nps_path = /n/gravy1/work/zyda/npspanel in the current release.

The resulting file 'user_name' may be executed.

Figure 3.5 NPSPD Source Code Compilation

IV. NPSPD DESIGN TOOLS

A. Panel Manager

The Panel Manager enables the user to interactively customize workspace panels. This tool is opened either by pressing the F8 key or selecting Panel Manager from the pull down menu. Figure 4.1 is an example of the Panel Manager window.

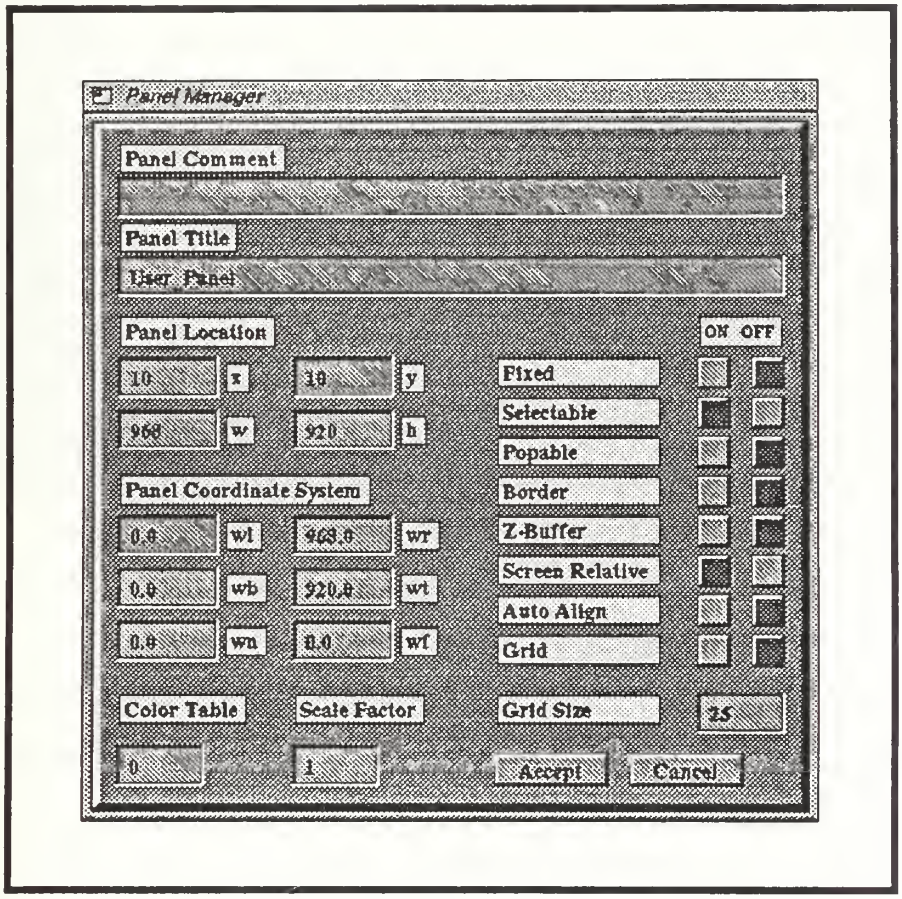

Figure 4.1 Panel Manager

The first typein across the top of the window is used to attach a comment to the panel. This comment will be saved in the intermediate file when the workspace is saved. The

second typein is used to change the title of the panel. Changes to this field will be reflected in the title bar of the workspace that is being edited.

The next group of typeins on the left side of the window are used to set the location and size of the panel. Changes to any of these parameters are immediately reflected in the panel. Below the panel location inputs are six typeins that are used to modify the world coordinates of the panel. These values only take effect if the panel is drawn in Screen Relative mode. Across the bottom of the window are three typeins that enable the user to set the panel's color table, scale factor and grid size.

On the right side of the window are nine sets of radio buttons. These buttons, which can be either ON or OFF, are used to set various flags for the panel. Refer to the User's Manual for a complete explanation of each flag and its meaning.

Finally in the bottom right corner of the window are two buttons. The Accept button is used to make any changes to the panel's parameters permanent. The Cancel button is used to undo any changes made to the panel in the current editing session and restore it to its previous state. Pressing either of these buttons completes the panel editing session and closes the window.

B. Actuator Manager

The Actuator Manager enables the user to interactively customize actuators. This tool is opened either by pressing the F9 key or selecting Actuator Manager from the pull down menu. Figure 4.2 is an example of the Actuator Manager window.

The first typein across the top of the window is the actuator comment field. Comments entered in this typein will be saved in the actuator's permanent comment field in the intermediate file when the actuator's host panel is saved. Below the comment typein is the label typein. This field is used to specify the label for the actuator.

Directly below the label typein are two buttons. The first is marked Label and it is used to control the location of the label string. The second is marked Value and it controls the location of the value output string. The position of these strings is determined by selecting one of the 16 position buttons directly below these two buttons. The 13 relative position buttons surrounding the box are defined as default positions. If a fixed postion is desired, either the Fixed button or the Fixed - Center button is selected. The fixed position is then

set by entering the appropriate x and y coordinates in either the Label Location typeins or the Value Location typeins.

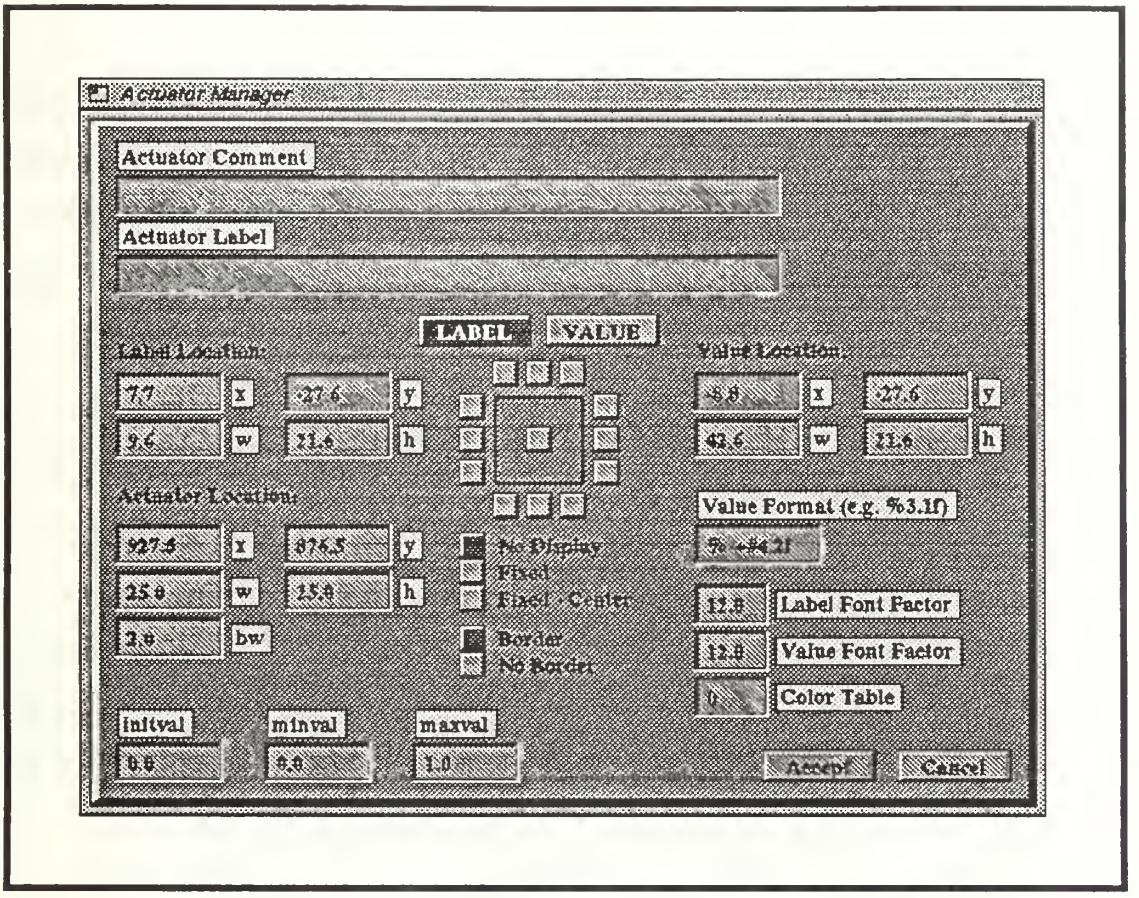

Figure 4.2 Actuator Manager

The actuator's position and size are set with the Actuator Location typeins. The initial, minimum and maximum values associated with the actuator are set with the appropriate typeins in the lower left side of the window.

The format of the value output string is set by entering the appropriate Unix format string in the Value Format typein. The font factor for the label and value strings is set with the Label and Value Font Factor typeins, respectively. Finally, the color table for the actuator is set with the Color Table typein.

The Accept button in the lower right side of Figure 4.2 is used to make any modifications to the actuator permanent. The Cancel button is used to undo any changes

made to the actuator in the current editing session and restore it to its previous state. Pressing either of these buttons completes the editing session and closes the window.

C. Color Manager

The Color Manager enables the user to interactively customize colors for actuators and their host panels. This tool is opened either by pressing the F10 key or selecting Color Manager from the pull down menu. Figure 4.3 is an example of the Color Manager window.

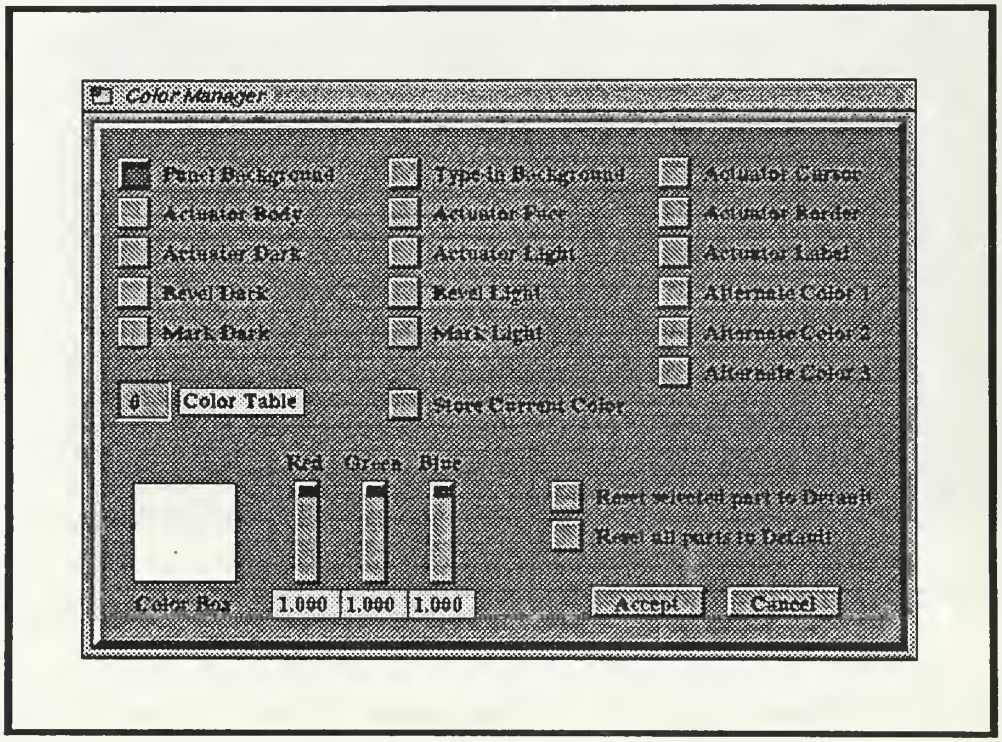

Figure 4.3 Color Manager

The NPSPD allows users to define up to eight custom color tables. Within each color table are 24 pre-defined panel and actuator colors. The first eight colors in the table are the basic colors, such as black, white, red, etc. These colors can not be changed by the user. The remaining 16 colors, defined as Panel Background, Actuator Body, etc. can be modified using the Red, Green and Blue sliders. As these sliders are moved, the resulting RGB color is displayed in the Color Box in the lower left corner of the window. The

corresponding color in the actuator or panel is also drawn, if applicable. When the desired color is obtained, pressing the Store Current Color button will make the modification permanent. This must be done for each modified color. Colors can be restored to their default values at any time using the two Reset buttons as appropriate. The functionality of the Accept and Cancel buttons is the same as the Actuator and Panel Managers.

D. Intermediate File Manager

The Intermediate File Manager tool enables the user to save and recall panel designs. Refer to Chapter VI and Figure 6.1 for a description of its use.

E. Source Code Manager

The Code Manager tool enables the user to generate source code that corresponds to an interface design. Refer to Chapter VII and Figure 7.1 for a description of its use.

F. Information Manager

The Information Manager displays to the user various messages during the NPSPD session. It is opened by the system when an action by the user either causes an error or can not be completed. It is closed by pressing the Continue button. Figure 4.4 is an example of the Information Manager window.

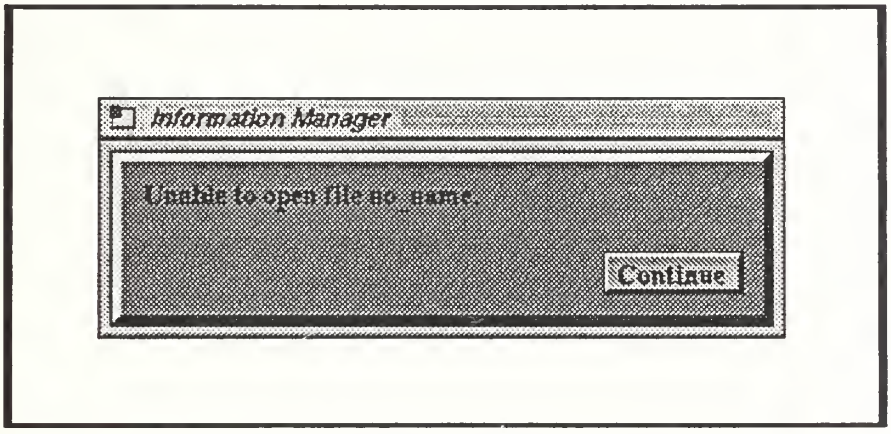

Figure 4.4 Information Manager

G. Help Manager

The Help Manager provides the user on-line NPSPD manual pages. This tool is opened by pressing the F1 key. Figure 4.5 is an example of the Help Manager window.

The desired set of manual pages is selected by pressing the appropriate button. The user can scroll through the text using either the up and down arrow buttons or the scroll bar on the typeout. The Help Manager window is closed by either pressing the F1 key or the Close key.

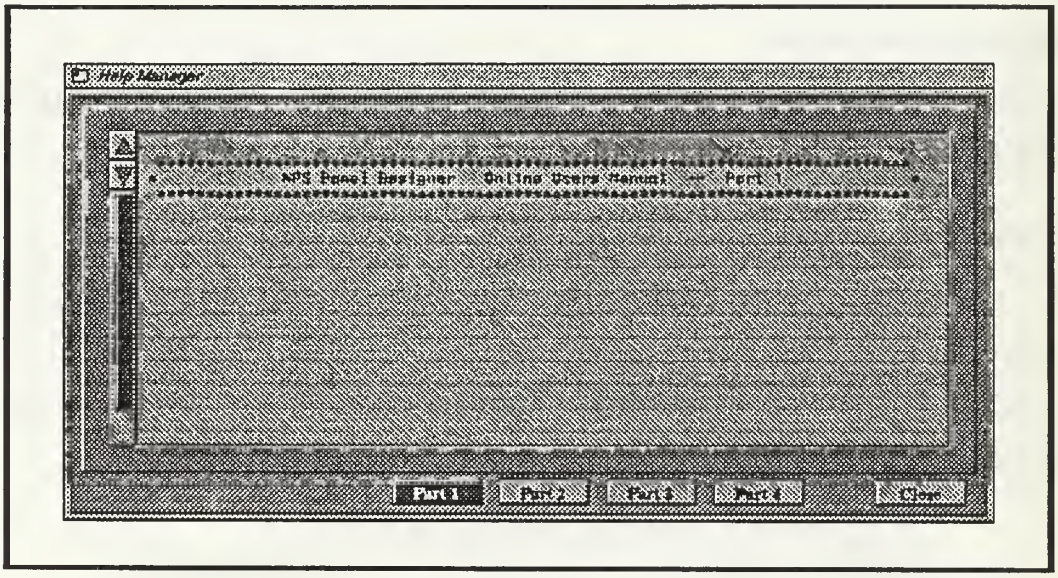

Figure 4.5 Help Manager

V. NPSPD TOOLBOX LIBRARY

The NPSPD ToolBox provides a library of panel and actuator structures with the access and control functions necessary to implement graphical user interfaces. The ToolBox is designed so that default settings for the panels and actuators are sufficient to build a basic interface. Modifications tailor the interface to the needs of the application. This chapter describes the contents of the ToolBox including examples of programming level use. The NPSPD Reference Manual provides a complete description of the Panel ToolBox and its use. Figure 5.1 presents an example of the creation and modification of a panel with a single Dial actuator.

```
{
    Panel *p;                          /* Temporary panel pointer          */
    Actuator *a;                       /* Temporary actuator pointer       */

    p = create_panel ();
    set_panel_location(p, 20, 56);
    set_panel_size(p, 720, 534);
    set_attribute(p, visible, TRUE);
    set_attribute(p, fixed, FALSE);
    set_panel_title(p, "User_Panel");
    set_attribute(p, color_table, 1);
    append_panel(p, Panel_List);

    a = create_actuator(dial);
    set_actuator_location(a, 77.5, 119.5);
    set_actuator_size(a, 75, 75, 2);
    set_actuator_label(a, BOTTOM, 10, "Object Rotation Control");
    set_attribute(a, activefunc, rotate_object);
    set_detail(Dial, a, major_tics, 4);
    set_detail(Dial, a, minor_tics, 1);
    set_detail(Dial, a, winds, 1);
    set_detail(Dial, a, finefactor, 0.1);
    insert_actuator(a, p);
}
```

Figure 5.1 Creation and Modification Example

A. Initialization Procedures

The Panel ToolBox requires several initialization steps to ensure proper operation. *Initialize_ToolBox()* sets up the ToolBox environment, initializing global state variables,

panel management linked lists, the event queue, keyboard buffers, color tables, and fonts. Panel and actuator creations and modifications follow. There is no initialization constraint on either panels or actuators except that the host panel for each actuator must exist before that actuator may be added. Figure 5.2 presents the initialization code generated by NPSPD.

```
void initialize_main()                    /* initialize panel environment          */
{

    initialize_ToolBox();                 /* initialize NPS Panel ToolBox          */

    initialize_panels();                  /* Initialize the control panels         */

    initialize_actuators();               /* create the actuators                  */

    initialize_colors();                  /* initialize user defined colors        */

    /*----- initialize all other aspects of main program.                          */

    user_init_queue();                    /* initialize event graphics queue       */
    user_init_menu();                     /* initialize PanelDesigner menus        */
    user_init_cursor();                   /* initialize special cursors            */
    user_init_overlay();                  /* initialize overlay planes & color     */

    /*----- User define initializations are called via user_init_main.             */

     user_init_main();                    /* user defined main initializations     */
}
```

Figure 5.2 NPSPD Initialization Sequence

B. Creation Procedures

The Panel ToolBox provides two functions for creation of default panels and default actuators. *Create_panel()*, which requires no arguments, allocates and initializes a panel data structure. *Create_actuator()*, requires an initialization function as its single argument and allocates an actuator basic data structure and unique detail structure as required by the initialization function. Both create functions return a pointer to the new object. Table IV presents a list of the initialization functions that may be used as an argument for *create_actuator()*.

basic	dirview	scroll
box	fileview	slider
buffer_act	frame	vbar_slider
button	list_act	vstrip_slider
simple_button	listview	hbar_slider
toggle_button	menu	hstrip_slider
radio_button	arc_meter	slideroid
arrow_button	filled_arc_meter	stripchart
double_arrow_button	dial_meter	dual_stripchart
label_button	filled_dial_meter	hstripchart
cycle	vbar_meter	vstripchart
dial	vstrip_meter	title
square_dial	hbar_meter	typein
round_dial	hstrip_meter	typeout

Table IV ToolBox Actuator Initialization Functions

C. Insertion Procedures

Once a panel is created, it must be inserted into P*anel_List*, the linked list of panels maintained by the ToolBox. *Insert_panel()* places the new panel at the head of the list. *Append_panel()* places the new panel at the tail of the list. The order of *Panel_List* determines the order of panel processing and display. The linked list is traversed from head to tail.

Likewise after an actuator is created, it must be attached to a panel or in some cases to a parent actuator. *Insert_actuator()* and *append_actuator()* add the new actuator to a panel's actuator list, at the head and tail respectively. *Add_sub_actuator()* inserts a specified actuator into another actuator's sub-actuator list (*sa*). Sub-actuators are used by several compound actuators including the Dirview, Fileview and Frame.

D. Modification Procedures

The Panel ToolBox provides a broad compliment of functions for modifying the attributes and details of panels and actuators. The designer directly controls the appearance and function of an interface by way of these modification functions. Modifications may be made both before and after the panel or actuator is added to the interface. *Set_panel_location()* and *set_panel_size()* position and size a panel. *Set_actuator_location()* and *set_actuator_size()* position and size an actuator.

Set_minvalue() and *set_maxvalue()* set limits on the value range for an actuator. The NPSPD Reference Manual lists and discusses all of the ToolBox functions, their arguments and their use. Two other general modification functions, *set_attribute()* and *set_detail()*, are discussed below.

1. Set_attribute()

Each of the attributes maintained in a panel or an actuator base structure may be modified using the *set_attribute()* function. As depicted in Figure 5.1, the arguments for the function call are the panel or actuator pointer, the attribute field name (e.g., visible and activefunc), and the value to be assigned to that attribute. Although some attributes are normally accessed and set by specialized functions such as *set_actuator_size()*, they may also be set using the *set_attribute()* function. An exception applies to the string attributes, *title*, *label* and *value_fmt*. These attributes must be set using the specialized functions provided by the ToolBox, *set_panel_title()*, *set_actuator_label()* and *set_value_format()*.

2. Set_detail()

Set_detail() provides the means to modify actuator detail parameters. The function call requires four arguments: the actuator detail data-type, the actuator pointer, the detail field name (e.g., *major_tics* and *minor_tics*), and the value to be assigned to that detail field. A specialized string function, *set_detail_string()*, provides the means to set an actuator detail string field (e.g.,the Typein *buf* field).

3. Binding Modifications

Modifications made to panels and actuators may affect several other aspects of the object. *Fix_panel()* and *fix_actuator()* ensure that all inter-related aspects of the object are adjusted after modifications are completed. Fix functions are specific to each panel and actuator, and they are automatically executed by the ToolBox when any of the insertion functions are called. Normally modifications are made immediately following creation and prior to insertion, thus binding is automatically ensured by the ToolBox. However, modifications may be needed at other points in an application program, possibly in response to user actions. After changes to the attributes of a panel, *fix_panel()* should be

explicitly called, and after changes to the attributes or details of an actuator, *fix_actuator()* should be called.

E. Processing Cycle

A graphics application is normally structured with a main program loop that repeatedly calls several functions. These functions typically include input processing, followed by interface display update, followed by applications calculations and display update. Figure 5.3 presents the main function and processing support functions generated by NPSPD and supported by the Panel ToolBox.

Control of the interface consists of processing the mouse, keyboard and other device inputs in *process_program_queue()* and processing the interface panels and actuators based on those inputs in *process_panels()*. The ToolBox manages the necessary state variables for mouse position, button action and keyboard action. *Reset_ToolBox_Q()* and *process_ToolBox_Q()* manage the event queue with respect to the interface. Event tokens are also passed to the application program via *user_process_queue()*. *Process_panels()* manages the selected panel and selected actuator ensuring that actuator state and value reflect the user mouse and keyboard inputs.

F. Processing Techniques

The Panel ToolBox supports optional, developer defined action functions that are executed during user activation of a panel and/or actuator. Panels and actuators have three pointers that may be set to reference the developer defined functions. These three attributes are *downfunc*, *activefunc* and *upfunc*. If they are assigned application functions, *downfunc* executes once when the left-mouse button transitions down, *upfunc* executes once when the left-mouse button transitions up, and *activefunc* executes each time *process_panels()* is called in the application main loop. These function references provide a powerful control link for the interface developer.

The state of each panel and the state and value of each actuator is available to the application program. State testing functions including *is_active()*, *is_visible()*, *is_selectable()* and *test_flag()* return a Boolean result. State flags may be altered under

```
main()
{

    initialize_main();                      /* initialize main program          */

    forever {                               /* Panel main loop                  */

        control_program(Panel_List);        /* process controls and queue       */
        draw_control_panels(Panel_List);    /* draw user control panels & acts  */
        user_display();                     /* handle to call user functions    */
    }
}

void control_program                        /* Control program operation        */
(
    PanelList *panel_list                   /* specified panel list             */
)
{
    process_program_queue();                /* Process the graphics event queue */
    process_panels(panel_list);             /* Control panels based on user input */
}

void process_program_queue ()               /* Process graphics event queue     */
{
    short TOKdevice,                        /* Graphics event queue device token */
          TOKvalue;                         /* Graphics event queue token value */

    reset_ToolBox_Q();                      /* Prepare ToolBox for input process */

    while ( qtest() ) {                     /* Process all tokens available     */

        TOKdevice = qread(&TOKvalue);
                                            /* Standard ToolBox input processing */
        process_ToolBox_Q(Panel_List, TOKdevice, TOKvalue);

        switch( TOKdevice ) {               /* User Program specific Q processing */

        case RIGHTMOUSE:                    /* Right Mouse Controls Menus       */
        if ( TOKvalue == DOWN )             /* on TransitionDown process menu   */
        user_process_menu();                /* User defined menu processor      */
        break;

        } /* end switch */

    /*----- User defined queue function receives all TOKENs processed.          */

    user_process_queue(TOKdevice, TOKvalue);

    } /* end while qtest() */
}
```

Figure 5.3 NPSPD Processing Functions

application control using the *set_attribute()* function or the more specific *set_flag()* and *clear_flag()* functions. *Set_value()* and *get_value()* modify and access an actuator's value.

Special effects may be produced by selectively controlling a panel's or actuator's state, particularly the *visible* and *selectable* flags. *Set_panel_invisible()* and *set_panel_visible()* provide the means to build an effective multi-panel interface.

G. Display Considerations

The Panel ToolBox manages the display of all interface panels and actuators. Drawing occurs only when a change of state or value necessitates an update of the appearance. Actuators are drawn in the reverse order of the host panel's actuator linked list. Thus if two actuators overlap, the one inserted closest to the head of the panel's list is drawn on top.

The ToolBox provides eight modifiable color tables to support multi-color interface designs. Each panel and actuator references one of the color tables as specified by the *color_table* attribute. Changing the *color_table* index or directly modifying the color tables using *define_color_table()* under application control can produce useful effects in the interface.

H. Efficiency Considerations

The Panel ToolBox optimizes processing and drawing algorithms so as not to degrade real-time applications. Panels and actuators each have their own set of specific variables that allow them to be customized for a particular use. For example, panels can be designed so that they are only visible when they are needed, saving screen space and CPU cycles. Similarly actuators can be designed so that they are not selectable, effectively making them output devices (e.g., a Dial, which is normally an input device, can be configured to display the output of a function or operation).

A panel is re-drawn completely only when required by a move or re-size action. During other processing and drawing cases, only those actuators which have been altered and those which have been specifically designated for redraw are drawn. The ToolBox determines visibility and need for redraw at high levels within its hierarchical program flow and prevents excess low level processing when it is not required. The ToolBox processes only

the selected panel, if any. While processing panels, the selected actuator on each panel, if any, and the actuators requiring automatic processing are processed.

VI. NPSPD INTERMEDIATE FILE

The File Manager feature of the NPSPD, as shown in Figure 6.1, enables the user to save and recall workspace designs. This is done by writing all of the pertinent information for a workspace to an ASCII file that we call the intermediate file. This highly structured file enables the user to store and recall uncompleted work, combine two or more separate designs, and modify designs manually (outside of the NPSPD environment) by using any text-based editor.

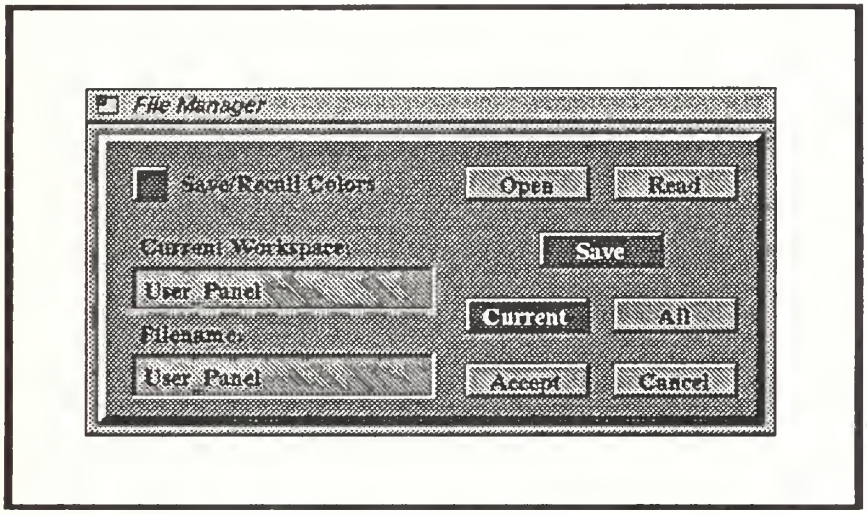

Figure 6.1 File Manager

A. INTERMEDIATE FILE LAYOUT

The intermediate file contains all of the information necessary to re-create a workspace(s) designed with the NPSPD. The basic layout of the file consists of panel parameters followed by actuator parameters for each panel, with at least one panel structure and zero or more actuator structures required as a minimum. The intermediate file also can be used to save and recall customized color table information.

1. REGULARITY

The layout of the intermediate file follows as closely as possible the structure of the type definition, with basic parameters listed first followed by zero or more lines of type-specific detail. Related parameters, such as location and size, are logically grouped together to improve readability. All panels have the same basic information listed with no detail necessary (panels have no type-specific detail). Actuators have the same basic information and detail based on the type. A simple actuator type, such as a box, has very little detail, while complex types, such as dials, have considerably more. The number of parameters needed to accurately re-create a panel or actuator is smaller than the complete set as specified in the type definition. This is because any of the variables in the type definitions are related only to the editing environment and can be set with defaults when panels and actuators are created.

2. SYNTAX

Figure 6.2 contains an example of an intermediate file produced by the NPSPD that contains one workspace and two actuators. The language for NPSPD intermediate files is case-insensitive, and white space is ignored.

a. File Header and Footer

The first line in the file is the reserved word header "Panel_Designer_File", identifying the file as an NPSPD intermediate file. During a read operation, if this line is not found by the parser, the file will be closed, the operation will be aborted and an informative message will appear. The last line in the file contains the reserved word "File_End". This signifies the end of the file.

b. Panels

Each panel saved to the intermediate file consists of a panel header containing the reserved word "Panel" followed by the title of the workspace, the parameters for the panel, zero or more actuator structures, and a panel footer consisting of the reserved word "Panel_End". An NPSPD intermediate file can have one or more panels.

```
Panel_Designer_File

Panel Box_Dial
/C This is an example of an optional permanent comment line C/
/* panel x, y, w, h */                          10 10 980 700
/* auto_align, grid_on, grid_size */            0 0 25.0
/* visible, selectable, fixed, popable */       1 1 0 0
/* border, screen_relative, zbuffer */          1 1 0
/* wl, wr, wb, wt, wn, wf */                    0.0 980.0 0.0 700.0 0.0 0.0
/* scale_factor, color_table */                 1.0 0

  Actuator BOX
  /* type, group_id, key_equivalent */          10 -41 0
  /* active, visible, selectable*/              0 1 1
  /* x, y, w, h, bw */                          469.5 208.5 85.0 25.0 0.0
  /* color_table */                             0
  /* l_location, label, label_font */           -13 "Box" 12.0
  /* lx, ly, lw, lh, lbx, lby */                24.2 1.7 36.6 21.6 4.8 5.8
  /* v_location, value_fmt, value_font, val */  0 "%-+#4.2f" 12.0 0.0
  /* initval, minval, maxval */                 0.0 0.0 1.0
  /* vx, vy, vw, vh, vbx, vby */                21.2 -27.6 42.6 21.6 4.8 5.8
  /* line_width, frgnd_clr, bkgnd_clr */        2 0 -1

  Actuator DIAL
  /* type, group_id, key_equivalent */          40 -23 0
  /* active, visible, selectable*/              0 1 1
  /* x, y, w, h, bw */                          671.5 402.5 75.0 75.0 2.0
  /* color_table */                             0
  /* l_location, label, label_font */           2 "Dial" 10.0
  /* lx, ly, lw, lh, lbx, lby */                22.5 -24.0 30.0 18.0 4.0 5.0
  /* v_location, value_fmt, value_font, val */  0 "%-+#4.2f" 12.0 0.0
  /* initval, minval, maxval */                 0.0 0.0 1.0
  /* vx, vy, vw, vh, vbx, vby */                16.2 -27.6 42.6 21.6 4.8 5.8
  /* mode, shape, r, major_tics, minor_tics */  2 1 33.8 4 0
  /* tl, tw, ml, mw */                          11.8 2.7 32.1 2.7
  /* theta, winds, finefactor */                0.0 1.0 0.1

Panel_End

Custom_Colors

File_End
```

Figure 6.2 Sample Intermediate File

The title for the panel must be a legal Unix file name. Following the title is an optional permanent comment line. Next are six lines containing all of the information needed to create a workspace. Following the panel information is the actuator data.

c. Actuators

Each actuator contains an actuator header and the parameters for the actuator. The actuator header consists of the reserved word "Actuator" followed by a reserved word identifying the actuator's basic type (Dial, Box, etc.). Next is another optional permanent comment line followed by nine lines of information basic to all actuators, including size and location, label string and location, and value location and format. Following this information are a variable number of lines containing detailed information unique to each type of actuator. For the first actuator in Figure 6.2, a Box, there is only one additional line. For the second actuator, a Dial, five additional lines are needed because of the added complexity.

d. Custom Colors

After the last line of the last actuator (in this example, the Dial) is a line containing the reserved word "Custom_Colors". The File Manager window contains a toggle button for saving and recalling custom colors (see Figure 6.1). If this option is selected, any colors that are modified during the current NPSPD session will be written to the intermediate file immediately after this line. When the file is read back in, the color tables in the working environment of the NPSPD will be modified with these colors.

e. Comments

The File Manager generates the same standard comments each time a workspace is saved. These standard C programming language comments desrcibe the parameters listed on each line of the file, making editing easier. They are discarded by the parser when the file is read. They can be modified and additional comments can be added anywhere in the file, but they will be lost the next time the file is read and then saved. Permanent comments that will be retained from session to session are allowed, but they are limited to one line in length, they must be bracketed by the characters "/C --- C/", and they must immediately

follow the title line for panels or actuators. The sample intermediate file in Figure 6.2 contains a permanent comment for the panel, but not for the actuators.

B. PARSER

A syntax-directed, recursive descent parser is used to read the intermediate file. The parser reads and acts on tokens created by a lexical analyzer while moving through the file recursively, following the syntax described above. It has a limited error analysis capability in that if it encounters an unexpected token, it will immediately abort the read operation and report to the user the general area of the problem. The parser currently has no error recovery capabilities, and we recommend this be added in future revisions.

1. Lexical Analyzer

The lexical analyzer reads a line of the file at a time and breaks it down into blocks of character strings. Blank lines, standard C programming language comments and white space (tabs, spaces, carriage returns, line feeds, control characters) are discarded. Each string is defined by a token identifier: GEN_ID for alpha-numeric strings, T_DECIMAL_LITERAL for numbers, T_COMMENT_TEXT for permanent comments and T_STR_LITERAL for alpha-numeric strings enclosed in double quotes. If an illegal character is encountered, the token T_ERROR is generated internally, a warning message is sent to the console, the string is discarded and the next string is read.

2. Reserved words

When a GEN_ID token is generated by the lexical analyzer, a function is called to check if the string is a reserved word (file header, panel header, etc.). A very efficient hash function that is an implementation of the reserved word table uses the first letter of the string to enter an array holding all of the reserved words for the NPSPD. It then uses a predefined offset to check if the string is among those strings starting with the same first character. If the string matches a reserved word, its unique token identifier (T_FILE_HDR, T_PANEL_HDR, etc.) is returned, otherwise the general identifier GEN_ID is returned. Appendix C lists reserved words for the NPSPD.

3. Numbers

In the case of T_DECIMAL_LITERAL tokens, the lexical analyzer builds a string in the form of a number, including a leading '-' sign for negative numbers and a decimal point for decimal numbers. If a '+' sign is encountered and the next character is a digit, the '+' sign is read and discarded. Spaces within the digits of the number are not allowed.

4. Comments

Standard C programming language comment strings are discarded by the lexical analyzer. Permanent comments optionally associated with each panel and actuator, as denoted by a leading '/C' and a trailing 'C/', will return the T_COMMENT_STR token.

5. Errors

The tokens returned by the lexical analyzer are used to create panels and actuators, so the order in which they are received is important. If the proper order of tokens in the intermediate file is not maintained, the parser will fill fields in the panel or actuator structures with erroneous data, and then either have extra tokens or not enough tokens at the end of the file. For this reason error messages produced by the parser will not always indicate the exact location of the problem. However they will indicate that a problem exists if it is not immediately obvious. The result of parsing an erroneous intermediate file is unpredictable since the panel and actuator structures are created as they are read.

C. MODIFICATIONS TO VALUES

The layout of the intermediate file lends itself to easy modification. The majority of editing tasks can be done interactively using the Actuator, Panel and Color Editing features of the NPSPD. However some types of modification, such as rearranging the order of panels and/or actuators, must be accomplished using a text-based editor.

VII. NPSPD SOURCE CODE GENERATION

One of the most powerful features of the NPSPD is its ability to generate source code that corresponds to an interface design. Using the Code Manager, the user is able to generate source code for the current workspace or all workspaces, and then modify that code using clearly defined entry points. The modified code can then be compiled and linked with an application, providing a custom interface in a minimum amount of time.

A. Code Manager

The F12 key in the NPSPD opens the Code Manager, as shown in Figure 7.1. The Current Workspace typein contains the title of the workspace that was current when the F12 key was pressed. Any valid workspace title can be entered in this field. The default generation mode is for the current workspace only. This can be changed by pressing the appropriate button (either Current or All). If Current is selected, code will be generated for the workspace corresponding to the Current Workspace field. If All is selected, code will be generated for all of the workspaces on the screen regardless of the contents of the Current Workspace field. The name of the output files can be any legal Unix file name, and does not have to be the same as one of the workspaces.

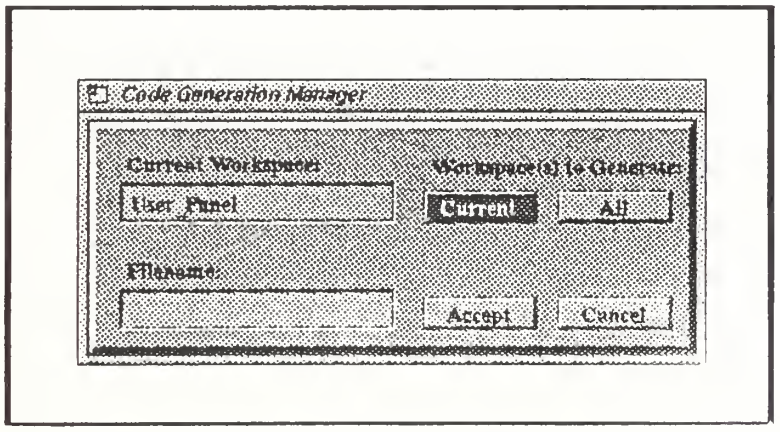

Figure 7.1 Code Manager

B. Generated Code

The NPSPD Code Generator generates three files that are then modified for use in an application. If the title of the file name in the Code Manager is 'User_Panel', the three generated files will be User_Panel.c, User_Panel_fn.c, and User_Panel.h. The first file contains all of the functions needed to initialize, display and control the user defined panel(s) created within NPSPD using the NPSPD ToolBox. The second file contains the user modifiable functions needed in support of the panel(s) generated by the NPSPD. The third file is the header file for the two source files. Refer to Appendix D for a complete listing of these three files. Refer to the NPSPD User's Manual in Appendix A for detailed instructions on modifying the generated code.

1. User_Panel.c

This file is complete as generated, but can be modified by the user. The most likely area of modification is in the creation of the panels and actuators in *initialize_panels()* and *initialize_actuators()* and the modification of colors in *initialize_colors()*. Any parameter for a panel or actuator can be changed, taking care to maintain the proper structure and function. Figure 7.2 is an example of the main loop in User_Panel.c.

```
main()
{
    initialize_main();                  /* initialize panel environment          */
    forever {                           /* Panel main loop                       */
     control_program(Panel_List);       /*  process controls and queue           */
     draw_control_panels(Panel_List);   /* draw control panels & actuators       */
    /*----- User designed calculations and 2D or 3D drawing functions are        */
    /*----- accessed via user_display(); User must manage any extra              */
    /*----- windows required                                                     */
     user_display();                    /* handle to call user functions         */
    }
}
```

Figure 7.2 User_Panel.c main loop

2. User_Panel_fn.c

This file is where the user ties the interface to the application. The template of the file provides the user entry points to initialize all aspects of the interface and the application, process the event queue, and process and display the application itself. Figure 7.3 at the end of this chapter presents an example of User_Panel_fn.c before modifications.

a. user_init_queue()

This function allows the user to add queue devices to the standard devices initialized by the NPSPD (e.g., RIGHTMOUSE and ESCKEY). Refer to the Reference Manual for information on additional devices queued in the NPSPD ToolBox.

b. user_init_menu()

The NPSPD generates a standard generic menu. This function allows the user to customize the menu system.

c. user_init_cursor()

This function allows the user to customize the cursor for the application.

d. user_init_overlay()

This function allows the user to customize the overlay planes for the application.

e. user_init_main()

This function is called from the function *main()* in User_Panel.c and allows the user to perform initializations specific to the application.

f. user_process_queue()

This function is called after the ToolBox has processed the queue and allows the user to do any additional processing of queue event tokens.

g. user_process_menu()

This function allows the user to process the menu system.

h. user_display()

This is where the user will do the bulk of the processing for the application. It is called each time through the drawing loop after the processing of the ToolBox control functions is completed.

i. user_exit()

Customized application exit procedures are placed in this function.

j. Entry Point Modification

Each of the source code entry points can be modified individually. However, in some cases, changes to one function require changes to one or more others. For example, if *user_process_queue()* refers to a non-standard queue device, that device must first be queued in *user_init_queue()*. Each function is clearly documented and provides the designer guidelines for modifying values or structures.

3. User_Panel.h

This header file contains the forward function declarations for all of the functions in the file User_Panel_fn.c, as well as the arrays holding the control panels and actuators and global variables and constants. Any application files that reference these functions and/ or variables must include this file.

C. Compiling and Linking

Instructions for compiling and linking these files are included at the top of each file. Refer to Chapter VIII for a detailed example of this procedure.

```
/*****************************************************************************
*  File:     User_Panel_fn.c      User defined calculations and               *
*  Version:  1.0                  drawing functions                           *
*  date:     90/12/01                                                         *
*  Author:   Richard M. Prevatt                                               *
*            David M. King                                                    *
*  ------------------------------------------------------------------------- *
*  Notes:                                                                     *
*        90/08/13  Created.                                                   *
*  ------------------------------------------------------------------------- *
*  This file contains the User modifiable functions needed in support of      *
*  the control panel generated by PanelDesigner.  Changes and additions may   *
*  be added to all files taking care to manage any extra windows.             *
*  It is used in conjunction with User_Panel.c                                *
*                                                                             *
*  The actual name of this and the related files was derived from the         *
*  name of the current workspace when it was produced by PanelDesigner.       *
*  { Substitute the actual name for 'User_Panel' in these instructions. }     *
*                                                                             *
*  If a file by that name already existed, the PanelDesigner saved the        *
*  the original version in a backup file as follows:                          *
*    User_Panel_fn.c  -->  User_Panel_fn.c.bak                                *
*                                                                             *
*  Compile as follows:                                                        *
*                                                                             *
*        cc -o user_name User_Panel.c User_Panel_fn.c /nps_path/lib/npspanel.a *
*                  /nps_path/include -O2 -align16 -G 0 -lc_s -lgl_s -lfm -lm  *
*                                                                             *
*        /nps_path must be defined as the proper path to the NPS Panel ToolBox.*
*        /nps_path = /n/gravy1/work/zyda/npspanel in the current release.     *
*                                                                             *
*  The resulting file 'user_name' may be executed.                            *
*****************************************************************************/


#define     EXTERN   extern      /* declarations are external              */
#define     INIT(x)              /*   and not initialized here             */

#include    "gl.h"               /* Graphics Library declarations          */
#include    "device.h"           /* Device declarations                    */

#include    "tbx.h"              /* Panel Toolbox Declarations             */

/*---------------------------------------------------------------------------*/
```

Figure 7.3 User_Panel_fn.c before modifications

```
/* ------------------------------------------------------------------------------------------------ */
/*  User defined and modifiable constants and declarations                                          */

#include      "User_Panel.h"

/* ------------------------------------------------------------------------------------------------ */
/*  User modifiable function definitions                                                            */


/* ------------------------------------------------------------------------------------------------ */
void      user_init_queue()        /* User defined queue init                                       */
{
 /*..... Place user needed event queue device initializations here.                                 */

}


/* ------------------------------------------------------------------------------------------------ */
void      user_init_menu()         /* User defined menus here                                       */
{
 /*..... Place user defined menu initializations here.                                              */

 main_menu = defpup(" Sample Main Menu      %t ");
 addtopup(main_menu," Place menu choices here %x100 ");
 addtopup(main_menu," Quit Program          %x999 ");
}


/* ------------------------------------------------------------------------------------------------ */
void      user_init_cursor()       /* User defined cursor init                                      */
{
 /*..... Place user defined cursor initializations here.                                            */

}


/* ------------------------------------------------------------------------------------------------ */
void      user_init_overlay()      /* User defined overlay init                                     */
{
 /*..... Place user defined overlay initializations here.                                           */

}

/* ------------------------------------------------------------------------------------------------ */
```

Figure 7.3 User_Panel_fn.c, cont.

```
/*-------------------------------------------------------------------------------------------------------*/
void      user_init_main()        /* User defined main initializations                                  */
{
 /*..... Place user defined initializations here.                                                       */
 /*..... This is called after all panel and actuator setup initializations.                             */

}


/*-------------------------------------------------------------------------------------------------------*/
void      user_process_queue      /* User defined queue functions                                       */
(
 short     TOKEN,                 /*  Graphics event Q device token                                     */
 short     TOKvalue               /*  Graphics event Q token value                                      */
)
{
 /*..... Place user defined queue processing here.                                                      */
 /*..... All queued tokens will be passed to this function after they are                               */
 /*.....  processed by the Panel ToolBox functions.  They may be used by                               */
 /*.....  the User's program to specify additional actions, etc.                                        */

}


/*-------------------------------------------------------------------------------------------------------*/
void      user_process_menu()     /* User defined menu processor                                        */
{
 long      choice;

 choice = dopup(main_menu);

 switch ( choice ) {

  /*----- Include other menu selection processing here.                                                 */

  case MENU_QUIT:                 /* exit the program                                                   */
   user_exit();
   break;

  default:
   break;
 }

}
```

Figure 7.3 User_Panel_fn.c, cont.

```
/* ------------------------------------------------------------------------------------------ */
void      user_display()          /* All user calc & drawing functions                        */
{
 /*..... Place user defined calculations and display control here.                           */
 /*..... This is called during each drawing loop after control panel                         */
 /*.....  processing is completed.                                                           */

}


/* ------------------------------------------------------------------------------------------ */
void      user_exit()             /* Clean up and exit the program                            */
{

 /*..... Place user defined exit procedures here.                                            */

 panelExit();                    /* Clear and close all Panel windows                        */

}

/*****************************************************************************
*  EOF: User_Panel_fn.c      { lines:   85 }                                    *
 *****************************************************************************/
```

Figure 7.3 User_Panel_fn.c, cont.

VIII. COMPLETE NPSPD APPLICATION

Designing and implementing a user interface with NPSPD is a simple process that can take as little as 15 minutes. Chapter III discusses in detail the five basic phases that make up the development process: preliminary design of the interface content and layout, development of the interface in the NPSPD environment, generation of compilable source code, modification of the interface and application source code to include appropriate communication links, and finally, compiling and linking the NPSPD interface code with the application code.

A. Building an Interface

We have chosen the NPS Autonomous Underwater Vehicle simulator (AUV) as the example application. This simulator controls a submersible vehicle in three dimensions as it navigates in various bodies of water. The application's user interface currently includes standard pull-down menus and the IRIS Spaceball. The Spaceball controls the motion of the AUV, and its eight buttons toggle environment flags and other choices. Currently, the AUV simulator does not use the mouse.

The example interface to be designed will include both input and output actuators and will utilize the keyboard, the mouse and the Spaceball. It will have five basic panels: a viewing control panel, a button control panel, two instrument panels and a welcome screen. The panels will contain several different types of actuators, and one of the panels will be hidden unless called for.

1. Starting NPSPD

NPSPD is invoked by typing 'npspd' and requires no command line arguments. The user should ensure that the NPSPD ToolBox library is accessible by their application for compiling and linking.

Upon invocation NPSPD presents a welcome screen which remains visible while the Palette and workspace panels are being initialized, and closes when initialization is completed. The Palette of standard actuators is on the right side of the screen and the

workspace is on the left. The title bar for each panel reflects that panel's current title. The lightning bolt icon, normally on the right side of the title bar and used to close a panel, is not present. Figure 3.2 in Chapter III illustrates the opening layout.

2. Creating the Panels

The first step is to create the five panels. We position and re-size the first workspace to become the first panel. We create new workspaces by pressing the F5 key. The remaining four panels are placed using this function and the Panel Editor. The final panel layout will be three panels across the bottom of the screen, each approximately 250 pixels high with the two end panels 300 pixels wide and the middle panel 580 pixels wide. The fourth panel will be above the lower left panel and will be 400 pixels wide by 250 high. The fifth panel will measure 500 X 250 pixels and will be centered in the middle of the screen.

Next we want to set the environment tools for each panel. We want to draw an alignment grid in each panel, and we also want to have the actuators snap to the grid when they are placed or moved. To accomplish this we place the mouse in each panel consecutively and press the F2 (AutoAlign) and F3 (GridDraw) keys. The default grid size is 25 pixels. It may be changed if needed using the F4 key and menu to pick a standard size, or the Panel Editor to set any size. For now we'll leave all panels at 25 pixels. Checking the status box on the bottom of the Palette confirms that AutoAlign is now on for each panel and the grid size is 25.

3. Customizing the Panels

After all of the panels are placed and sized, we open the Panel Editor and customize the attributes for each panel. For all panels, we want to set the Fixed flag 'ON' and the Border flag 'OFF'. This will prevent the user from moving or resizing the panels. Next we want to name each panel with a descriptive title. In our interface, these titles will not be visible because the panels will be drawn without borders. However it is still advisable to do this for two reasons. First, if we later go back and turn the borders on, the title will be displayed. Second, when we save our work to an intermediate file, a title for each panel will help us to keep track of individual panels and their purpose. Finally we can

attach a comment to each panel for documentation purposes in the intermediate file. This is optional but also recommended. The initial layout of the panels is shown in Figure 8.1.

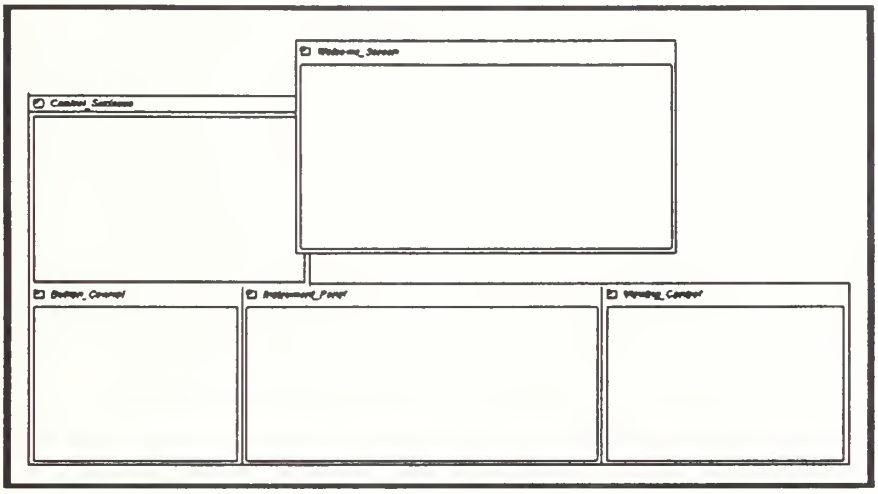

Figure 8.1 Initial Layout of panels for the AUV interface

4. Placing the Actuators

Each of the four panels contains unique actuators so we will step through their design separately. The design of some of the panels can be completed interactively, while others require that we leave the NPSPD environment in order to edit the intermediate file.

a. Viewing_Control Panel

The first panel is the Viewing_Control panel. This panel will have four actuators: three sliders that will control inclination, azimuth and distance, and a dial that will control the twist. To place the first slider, we move the mouse-cursor over the standard vertical slider on the Palette and press and release the middle-mouse button. This actuator then becomes the current actuator on the Palette as denoted by the surrounding white box. We then move the mouse to the location on the Viewing_Control panel where we want to place the lower-left corner of the slider and press and release the middle-mouse button again. This places a copy of the current Palette actuator (a standard vertical slider) on the panel. The slider will appear to jump, or 'snap' to the nearest grid intersection on the panel

because the AutoAlign tool is on. Whenever we place or move an actuator, the origin of the actuator (lower left corner) will move to the nearest grid intersection point, in this case a multiple of 25.

We need another vertical slider so we press the insert key on the keyboard and an exact duplicate of the current workspace actuator is created to the right of the original (alternatively we could have moved the mouse to the location of the second slider and pressed and released the middle-mouse button again to place a second copy).

To move an actuator on a workspace, we place the mouse over it, press and hold the middle-mouse button, and drag it to its new position. We now do this to the copy of the original slider we just created. Next we place a horizontal slider above the two vertical sliders using the same procedure. Finally we place a dial in the middle of the sliders (see Figure 8.2).

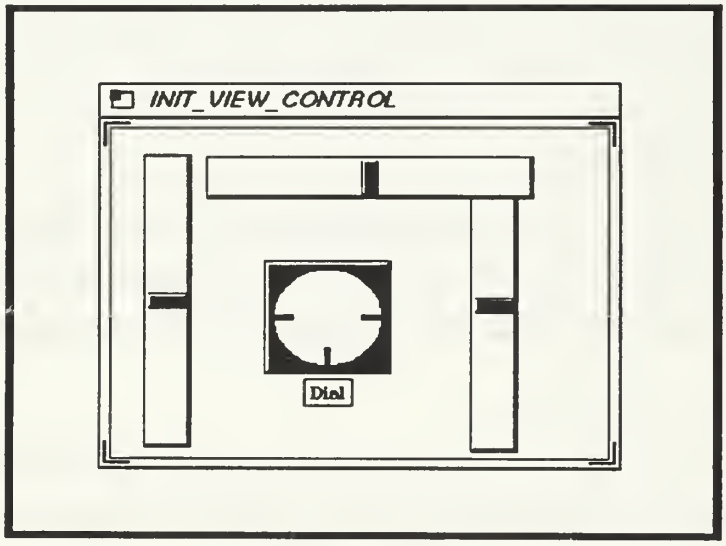

Figure 8.2 Initial Viewing_Control panel

Now that we have placed all of the actuators, we need to edit them. We set the left-most slider as the current workspace actuator and press the F9 key. This opens the Actuator Editor dialogue window. Using this tool, we will give the actuator a label, set its location and size, and attach a comment to it for the intermediate file.

First we will initialize the label. In the upper-left side of the dialogue window is a typein labeled "Actuator Label". We place the mouse over this typein and press and release the left-mouse button, activating the typein. The label of this actuator is "Inclination" so we type that in and press return. In the middle of the Actuator Editor is a representative box surrounded by nine small buttons with one in the center. We want to center the label under the bottom of the actuator so we place the mouse over the bottom center button and press and release the left-mouse button. The label now appears in the selected position. We want the label to have a white background so we won't change that. Next we change the font of the label from the default 10.0 point to 12.0 point by placing the mouse over the Label Font typein, activating it and entering the desired size.

We change the width and height dimensions of the slider from the standard 25 X 200 pixels to 20 X 150 by entering the new dimensions in the appropriate Actuator Dimensions typeins. Finally we add a comment to this slider to record its purpose in the intermediate file. We do this by entering a descriptive statement in the Actuator Comment typein at the bottom-left of the Actuator Editor.

If, after verifying all of the parameters for the actuator, we want to make them permanent, we place the mouse over the Accept button in the lower right of the window and press and release the left-mouse button. If we make changes to the actuator or open the window and decide to not make any changes, we can press Cancel instead, and the actuator will be restored to its original state. Figure 8.3 illustrates the final layout of the Viewing_Control panel.

b. Instrument_Panel

The next panel to design is the Instrument_Panel. This panel will have three different types of meters: vertical and horizontal strip meters, dial meters and an arc meter. We select the meter icon on the Palette and drop it in the appropriate panel. The actuator that appears is a standard arc meter. We have no way to interactively change this meter to a different type so we'll have to place it in its approximate final location and change the type in the intermediate file. We have the same situation with the remaining meters so we'll create standards by using the insert key to copy the first meter and place them in their approximate locations using the mouse. Our panel now contains ten standard meters,

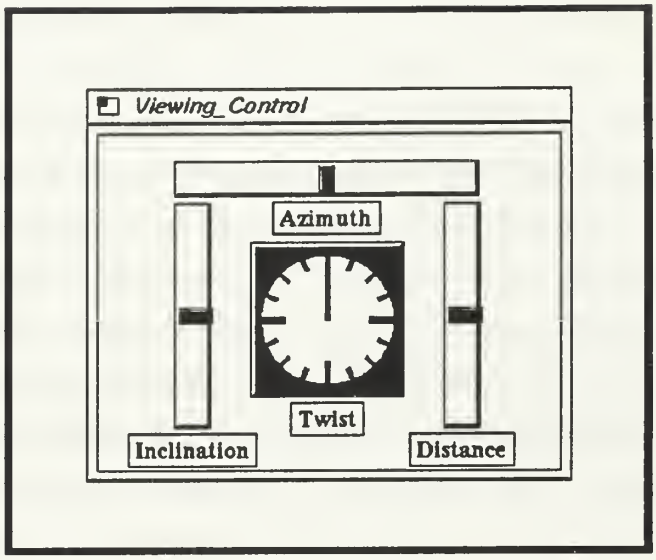

Figure 8.3 Final layout of the Viewing_Control panel

placed in their approximate final locations. Before we leave NPSPD to edit the intermediate file, we need to assign a label to each of the meters so that we can identify them later. We do this using the Actuator Editor. The meters will be labelled "Speed", "Pitch", "Depth", "RPM", Roll", "Heading", "Bow Rudder", "Stern Rudder", "Bow Planes" and "Stern Planes". For now we will place all of the labels below the actuators.

We now have to save our work and temporarily leave NPSPD environment so that we can edit the intermediate file. First we press the F11 key to open the File Manager window. We want to save all of the workspaces so we press the SAVE button and the ALL button. Next we need to enter a name for the file: this is an AUV interface so we enter AUV_panels. When we press return, the panels are saved and the window closes. Our workspaces have now been saved.

To edit the intermediate file, we exit NPSPD and open the file for editing using any text-based editor. Once in the file, we move to the panel labeled "Instrument_Panel" and locate the first actuator, which is the speed output. This is the meter that will display the speed of the AUV. We need to change its *mtype* detail field from 111 (METER_ARC) to 117 (METER_HBAR). Similarly we change the remaining meters

to their appropriate types. The modified intermediate file is illustrated in Figure 8.4 (only the first meter is listed). Refer to the NPSPD User's Manual in Appendix A for a complete description of actuator detail parameters and their modification.

```
Panel_Designer_File

Panel Instrument_Panel
/* panel x, y, w, h */                          424 9 522 230
/* auto_align, grid_on, grid_size */            0 0 25.0
/* visible, selectable, fixed, popable */       1 1 1 0
/* border, screen_relative, zbuffer */          1 1 0
/* wl, wr, wb, wt, wn, wf */                    0.0 522.0 0.0 230.0 0.0 0.0
/* scale_factor, color_table */                 1.0 0

 Actuator METER
  /* type, group_id, key_equivalent */          110 -240 0
  /* active, visible, selectable*/              0 1 1
  /* x, y, w, h, bw */                          15.5 177.5 123.0 16.1 2.0
  /* color_table */                             0
  /* l_location, label, label_font */           9 "Speed" 12.0
  /* lx, ly, lw, lh, lbx, lby */                0.0 22.1 50.6 21.6 4.8 5.8
  /* v_location, value_fmt, value_font, val */  7 "%-+#4.2f" 12.0 0.0
  /* initval, minval, maxval */                 0.0 0.0 1.0
  /* vx, vy, vw, vh, vbx, vby */                80.4 22.1 42.6 21.6 4.8 5.8
  /* mtype, r, major_tics, minor_tics */        117 31.9 1 1
  /* tl, tw, ml, mw */                          8.1 2.0 8.1 4.0
  /* mcolor, display_limits, limits_fmt */      16 0 "%-+#3.1f"
  /* damping_factor, history_ndx */             1 0
.
.
.

Panel_End

File_End
```

Figure 8.4 Instrument_Panel intermediate file (Meters)

At this point, we could change other attributes of the meters, such as location and size, but this is more easily done in the NPSPD environment where the changes can be seen immediately. We can now verify the changes we have made and exit the file.

Re-entering NPSPD, we press the F11 key to open the File Manager window and press the OPEN button. In the Filename typein we enter "Instrument_Panel" and press

return. The four panels we created earlier open, with the Instrument_Panel now containing the correct types of meters. We now need to customize each meter, setting its size and location, specifying a value location and format if necessary, and adding a comment. We select the first actuator, the "Speed" meter, and press F9 to open the Actuator Editor window. The label string is correct so we'll leave that. The location of the label and its font size need to be changed however, so we'll do that. We want to place the label above the meter and left-align it, so we press the top-left button on the location square. Next we want the label to be 12 point font instead of the default 10 so we enter the new font size in the Label Font Factor typein. We want a value to be displayed for this meter so we press the VALUE button near the top center of the window. For this meter, we want to place the value on the top of the meter and right-align it, so we press the top-right button on the location box. The default font for values is 12 point, so we'll leave that. However we need to change the value format string in the upper right of the window from the default "%-+#7.1f" to "%6.1f", which will give us a floating point number with six digits of precision and 1 digit to the right of the decimal point. Next we need to change the size and location of the meter. Using a rough sketch, we determine we want this meter to be 150 pixels wide by 20 pixels high, and the reference point should be at 14, 180 (x & y). Using the Actuator Location typeins, we enter these parameters. Finally, we add a comment for this meter, describing it as the speed output meter.

The remaining actuators on this panel are modified in the same fashion. Figure 8.5 illustrates the final layout of the Instrument_Panel.

c. Button_Control Panel

Before we begin placing actuators, we can make an adjustment that will save us some time. The current grid size for the Button_Control panel is 25 pixels. We want the buttons that we place on this panel to be 15 pixels apart. If we change the grid size to 5, it will be much easier to place the buttons exactly where we want them using the mouse rather than explicitly entering the coordinates. We can change the grid size for the panel using the Panel Editor and the Grid Size typein. As we do this, the grid on the Button_Control panel and the status box on the Palette reflect the change.

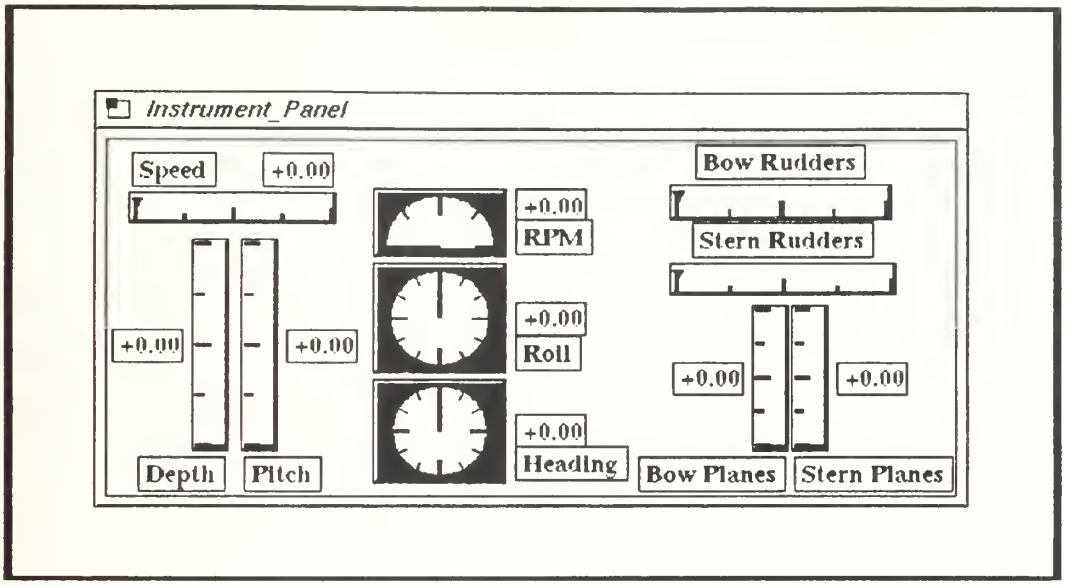

Figure 8.5 Final Layout of the Instrument_Panel

The standard toggle buttons are on the top-left side of the Palette. We have a choice of either square or round. We select the square button and place it on the Button_Control panel. We want the first button to be located at 20, 200. We can either use the mouse to move the button or open the Actuator Editor and enter the location explicitly. Since the grid size allows us to move 5 pixels at a time, using the mouse is probably faster, so we move the button to its final position using the mouse. The default size of buttons is 25 X 25 pixels, which is the size we want, so we don't change that. We need five more standard toggle buttons so we place the remaining copies in the appropriate locations: we need two columns of five buttons each, with 15 pixels separating buttons in the y direction and 170 pixels in the x direction. Column one (x = 20) will consist entirely of standard toggle buttons, while the first two and last two buttons in the second column (x = 190) will be radio buttons and momentary-action buttons, respectively. The middle button in the second column will be a standard toggle button. Accordingly we move the six standard buttons to their positions.

The radio buttons are placed next. Select the square radio button from the Palette and place it on the top of the right column (y = 200) on the Button_Control panel.

When a radio button is placed from the Palette, two copies will be created, so we move the second button down to the second position in the column using the mouse.

Finally we need two momentary-action buttons. These are located in the cross on the top of the Palette. We don't need a symbol on our buttons so we select the middle button in the cross and place copies in the fourth and fifth positions in the second column on the Button_Control panel.

Now that our buttons are in place, we need to edit them. Each button will have a label on its right side using a 12 point font factor, and each should have a descriptive comment. Using the Actuator Editor, we edit each button, entering its label and a comment. The final layout for the Button_Control panel is illustrated in Figure 8.6.

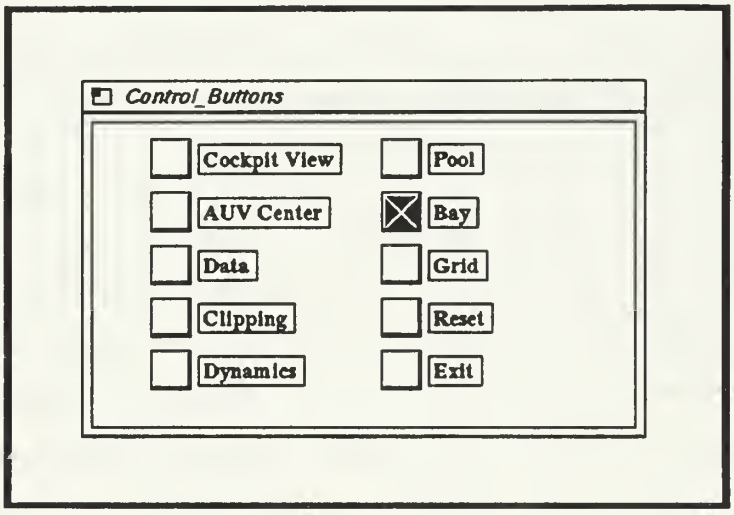

Figure 8.6 Final layout of the Button_Control panel

d. Control_Surfaces Panel

The next panel to design is the Control_Surfaces panel. This panel will contain 4 stripcharts, one each for the bow and stern rudders and the bow and stern planes. We select the Stripchart icon from the Palette and place the first stripchart at 25-25. The remaining three are placed at 25-150, 250-25, and 250-150. Opening the Actuator Editor, we enter a label for each stripchart, positioning it on the top center of the actuator. We want a value to be displayed on the right side of each stripchart so we press the VALUE button

on the editor and then the right middle button on the location box. We also need to change the value format string to "%7.1f" for each stripchart. Finally we add a comment to each one. We don't want to display the limits for the stripcharts but we'll have to change that in the intermediate file. Once we are done editing what we can on the stripcharts, we save our work (we can re-use the file name AUV_Panel) and exit NPSPD.

Using an editor to open the intermediate file, we move to the Control_Surfaces panel and locate the first stripchart actuator. First we change the *display_limits* flag from 1 (TRUE) to 0 (FALSE). Next we change the *Bind_High* and *Bind_Low* flags from 0 to 1. Finally, we change the *minval* and *maxval* values to -40.0 and 40.0, respectively. We make these changes to all four stripcharts. The Final layout of the Control_Surfaces panel is shown in Figure 8.7.

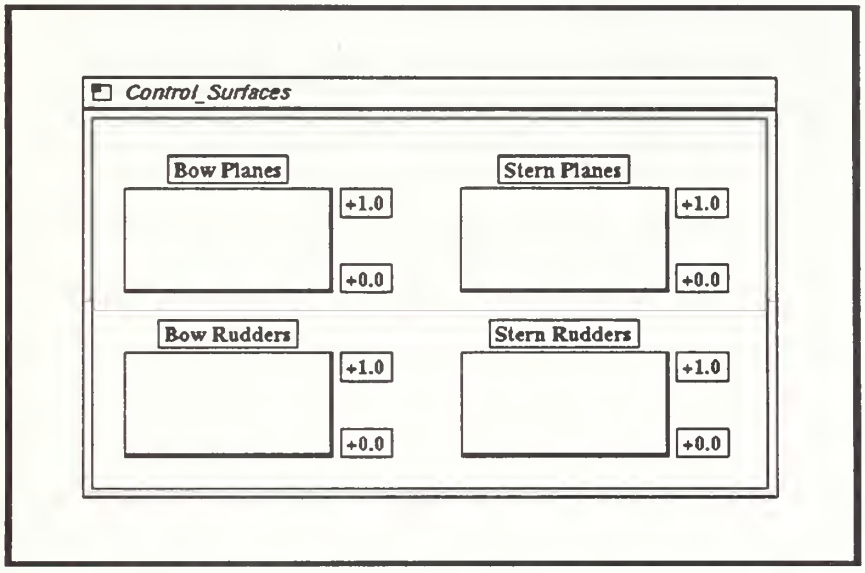

Figure 8.7 Final layout of the Control_Surfaces panel

e. Welcome_Screen Panel

The final panel to create is the Welcome_Screen. This panel will consist of two Title actuators, and will only be visible in our application momentarily while the data is initialized. We select the Title actuator from the Palette and place two copies on the Welcome_Screen panel, one above the other. Opening the Actuator Editor we enter the

label "NPS AUV" for the upper actuator. Next we change the width and height to 230 and 50, respectively. We then change the location of the actuator to 140, 180. Finally we change the Label Font Factor to 28 point. The second Title actuator is modified in the same fashion. The final layout of the Welcome_Screen is shown in Figure 8.8.

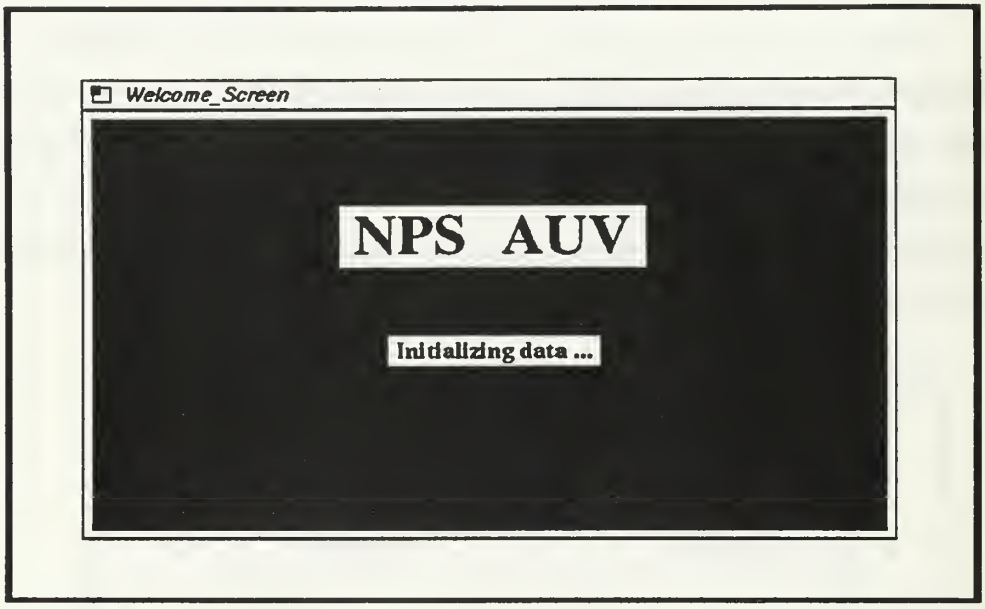

Figure 8.8 Final layout of the Welcome_Screen panel

B. Generating Code

Once we finish creating and editing the panels and actuators, we generate the source code that will be integrated into the AUV application. We open the Code Manager window by pressing the F12 key. First, we want to generate code for all five panels so we press the "All Workspaces" button. Next, we have modified at least one color table so we also press the "Save/Recall Custom Colors" button. Finally, we enter the name of the file to hold the generated code. We enter "AUV_Panel_Code" and press return. Three files have now been generated: AUV_Panel_Code.c, AUV_Panel_Code_fn.c and AUV_Panel_Code.h. The next step is to edit these files, compile them and link them with our application.

C. Editing the Generated Source Code

For our application, the only file we need to edit is "AUV_Panel_Code.c". This file contains the function calls to create the panels and actuators, the ToolBox initialization calls, and the user interface main control loop. First we will verify the panel and actuator creation calls.

1. Verifying the Panel and Actuator Creation Calls

The code generated for the creation of each panel consists of the following default lines: panel location and size, visible flag, fixed flag, border flag, popable flag, and the title for the panel. We specified the values for all of these parameters except the visible flag interactively earlier when we were in the NPSPD environment. For our application we only need to change one parameter on one panel. We only want the Control_Surfaces panel to be visible at certain times so we set its visible flag to FALSE. This means that when the application starts all panels except for Control_Surfaces will be visible.

2. Customizing the Code

Next we need to edit the function *main()*. Since we are integrating the NPSPD code into an existing application that has its own *main()* function, we need to rename ours. We thus change it to *unused_main()*. We will however have to move some of its calls to the existing *main()* function. We will do this in the next section.

D. Editing the Application Code

The application's header files and source code need to be modified in order to use the NPSPD interface. This can be accomplished using any text-based editor.

1. Header Files

The "globals.h" file in the AUV directory contains all of the global variable definitions and forward function declarations for the AUV simulator, and is included by all of its source files. This is where we include the NPSPD file "tbx.h", and the header file for the generated code, "AUV_Panel_Code.h". The file "tbx.h" contains the variable definitions and function declarations for the ToolBox library, and "AUV_Panel_Code.h" contains the variable definitions and function declarations for the interface code. When we

compile the AUV code, we will be linking to the NPSPD panel library so we can simply include these files as though they were local (in fact "AUV_Panel_Code.h" is local, but "tbx.h" is not).

Next we will define manifest constants for the names of the interface controls. This will make the AUV source code much more readable, thus making editing easier. In the file "AUV_Panel_Code.c" are all of our control structures in the form *Control[0][0]*, *Control[0][1]*, etc., and panel structures in the form *Control_Panel[0]*, *Control_Panel[1]*, etc. We want to define constants with more descriptive names that will be substituted for these variables in our AUV code. For example, we can reference the interface panels by using *Control_Panel[x]*, where x is the index of the desired panel, but then we have to remember the index number for all of the panels. By using descriptive names, such as VIEWING_CONTROL for *Control_Panel[0]*, we improve the readability of our code immensely and make editing much easier. The disadvantage of this is that if the order of the panels or actuators in the file "AUV_Panel_Code.c" changes we have to also change our definitions. But if we exercise a little care in modifying this file, we can avoid this problem. Figure 8.9 shows the modified "globals.h" file.

2. Modifying the Main Program

Since we are integrating the NPSPD interface into an existing application, we have to ensure several things happen when the application is initialized and each time through the control loop.

a. Initialization

The function *main()* in the AUV simulator makes a call to an initialization function, *initialize_auv()*. This function then makes calls to initialize each aspect of the AUV simulator: databases, devices, etc. We need to add several calls to this function to initialize the interface. These include: *initialize_main()*, *initialize_panels()*, and *initialize_actuators()*. The function *initialize_main()* initializes the interface panel environment, including default settings, colors, the event queue, the menus, the cursor and the overlay planes. Some or all of these calls may be unnecessary in the AUV simulator, but including them is a minor expense and ensures all necessary initializations are completed. The function *initialize_panels()* creates the interface control panels

```
/*        File: globals.h                                                    *
 * Description: Header file for NPS AUV Simulator                           */

/* General Include Files                                                   */
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#include <gl/spaceball.h>
#include "sys/types.h"

/* NPSPD & ToolBox Include Files                                           */
#include "tbx.h"
#include "AUV_Panel_Code.h"

/* NPSPD & ToolBox Manifest Constant Declarations - Panels                 */
#define VIEWING_CONTROL                 Control_Panel[0]
#define INSTRUMENT_PANEL                Control_Panel[1]
#define BUTTON_CONTROL                  Control_Panel[2]
#define CONTROL_SURFACES                Control_Panel[3]
#define WELCOME_SCREEN                  Control_Panel[4]

/* NPSPD & ToolBox Manifest Constant Declarations - Actuatorss             */
#define INCLINATION                     Control[0][0]
#define AZIMUTH                         Control[0][1]
#define DISTANCE                        Control[0][2]
#define TWIST                           Control[0][3]
.
.
.
```

Figure 8.9 Modified Globals.h File

(*Control_Panel[0]*, *Control_Panel[1]*, etc.). Finally *initialize_actuators()* creates the actuators on the control panels.

b. Main Control Loop

The AUV application has a function called *main_loop()*. This function contains the main control loop for the AUV simulator. While in a forever loop, this function looks for events on the queue, processes them and draws any changes. Figure 8.10 is a simplified code segment depicting the order of events in this loop.

```
main_loop(Sub_ptr auv)
{
  while(TRUE)                          /* loop forever                    */
  {
    if(qtest())
   {
       while (qtest())
       {
         sbdevice = qread(&value);     /* read spaceball                  */
         switch(sbdevice)
         {
           /* Spaceball functionality omitted ...                        */

           case REDRAW:
             reshapeviewport();
             break;

         } /* end switch */
       } /* end while qtest() */
    } /* end if qtest() */

    /* update the simulator values                                       */
    display_environment(auv);          /* contains OFF display routines   */
    swapbuffers();

  } /* end while(TRUE) */
} /* end main loop */
```

Figure 8.10 Original Main Control Loop for the AUV Simulator

The NPSPD interface needs to process queue events and draw any changes to controls each time through this loop. This can be done by adding five lines of code, as illustrated in Figure 8.11.

In order to process queue events so that the NPSPD interface can follow the action of the mouse, we need to add calls to reset the ToolBox queue and then process the ToolBox queue, in that order.We add our function *reset_ToolBox_Q()* to the top of the loop, just before we check if QTEST is true. This function resets mouse transition flags and other events that the ToolBox monitors. Next we add our call to process the latest queue event. We do this with the function *process_ToolBox_Q(Panel_List, sbdevice, value)*, inserted right after the queue event is read. This function takes the queue event and processes it according to where it occurred. For example, if the left-mouse button is

```
main_loop(Sub_ptr auv)
{
  while(TRUE)
  {
    reset_ToolBox_Q();

    if(qtest())
    {
      while (qtest())
      {
        sbdevice = qread(&value);        /* read spaceball                    */
        process_ToolBox_Q(Panel_List, sbdevice, value);
        switch(sbdevice)
        {
          /* Spaceball functionality omitted ...                             */

          case REDRAW:
            reshapeviewport();
            break;

        } /* end switch */
      } /* end while qtest() */
    } /* end if qtest() */

    /* update the NPSPD interface values                                     */
    update_panel_values(auv);

    /* update the simulator values                                           */
    display_environment(auv);        /* contains OFF display routines        */
    swapbuffers();

    /* process and draw the NPSPD interface controls                         */
    process_panels(Panel_List);
    draw_control_panels(Panel_List);

  } /* end while(TRUE) */
} /* end main loop */
```

Figure 8.11 Modified Main Control Loop

depressed on the Viewing_Control panel, the sbdevice will be the left-mouse, the value will be the transition down value of a mouse button, and the *process_ToolBox_Q()* function will run through the list of interface panels (*Panel_List*) and determine that the event occurred on the Viewing_Control panel. The *process_ToolBox_Q()* function does not alter the queue events, so any references made to these events subsequent to this call will be correct.

After the queue event is processed, we need to process the interface panels and their associated actuators. This is done by calling the function process_panels(Panel_List). This function runs through each interface panel (*Panel_List*) and processes the latest queue event. The AUV simulator interface is about evenly divided between input devices (sliders, dials, etc.) and output devices (meters, stripcharts, etc.). We chose to process the values of the interface control devices first, then update the AUV variables and interface. If the application interface consisted primarily of output devices, it would be more efficient to update their values first and then process the changes to the interface controls. However in either order the difference will be transparent to the user.

Next we need to update the status of the auv environment and the interface. This is done in a function called *update_panel_values(auv)*, where auv is a data structure containing the parameters for the vehicle and its environment. This function, illustrated in Figure 8.12, is called at the bottom of the main control loop, shown in Figure 8.11. Notice in some cases the auv variable takes the value of the interface control (e.g. auv->sys.Cockpit_view = COCKPIT_VIEW), while in other cases the opposite is true (PITCH = auv->pitch). In the first case the interface control is an input device, while in the second case it is an output device. All processing of controls and environment variables is done in this fashion.

The last thing to do in the loop is draw the changes to the interface panels and controls. This is done by calling the function draw_control_panels(Panel_List). This function checks each panel and actuator, determines if they need to be redrawn, and draws them, swapping buffers as necessary. If panels are not visible they are not processed or drawn, saving CPU cycles.

E. Linking the Application Code to the NPSPD Library

Once the interface and application code has been modified, we need to compile and link it. The AUV application utilizes a make file to do this. We need to modify it to include the ToolBox header file tbx.h, compile the interface code and link all of the object code to the NPSPD library. The modified Makefile is shown in Figure 8.13

```
void update_panel_values(Sub_ptr auv)
{

/* Interface Panel Input Devices (sliders, dials, & buttons)                    */

  auv->obs.inclination          = (Coord)INCLINATION;
  auv->obs.azimuth              = (Coord)AZIMUTH;
  auv->obs.twist                = (Angle)TWIST;
  auv->obs.distance             = (Coord)DISTANCE;

/* Interface Panel Output Devices (meters & stripcharts)                        */

  BOW_PLANES                    = auv->deflect[0];
  STERN_PLANES                  = auv->deflect[1];
  BOW_RUDDER                    = auv->deflect[2];
  STERN_RUDDERS                 = auv->deflect[3];
  RPM                           = auv->rpm[0];
  PITCH                         = (float)auv->pitch;
  ROLL                          = (float)auv->roll;
  HEADING                       = (float)auv->heading;
  DEPTH                         = (float)auv->depth;

  if(is_visible(CONTROL_SURFACES) {
    set_stripchart_value(Control[3][0], (float)auv->deflect[0]);
    set_stripchart_value(Control[3][1], (float)auv->deflect[1]);
    set_stripchart_value(Control[3][2], (float)auv->deflect[2]);
    set_stripchart_value(Control[3][2], (float)auv->deflect[3]);
  }
.
.
.
}
```

Figure 8.12 Update_Panel_Values Function

1. Including the ToolBox header file tbx.h

The file tbx.h is included by defining the variable INCLUDE to be the path to the NPSPD library, and adding the statement "-I${INCLUDE}" to the compilation line. This effectively makes that entire directory visible to the application's files.

2. Compiling the Interface Code

In order to compile the interface code (AUV_Panel_Code.c and AUV_Panel_Code_fn.c), we need to add their object file equivalents to the OBJS specification and their source file equivalents to the CODES specification. We also have to

define their dependencies, which in both cases is simply the corresponding source code and the AUV_Panel_Code.h header file

```
SHELL       =   /bin/sh
C           =   cc -I${INCLUDE}
LDCC        =   cc
CFLAGS      =   -g -O0 -w -G 0
LDFLAGS     =   -g
TBX         =   /n/gravy1/work/zyda/npspanel/lib/npspanel.a
INCLUDE     =   /n/gravy1/work/zyda/npspanel/include
LIBES       =   -lspaceball -lgl_s -lc_s -lmpc -lfm -lm
OBJS        =   dauv.o AUV_Panel_Code.o AUV_Panel_Code_fn.o ...
CODES       =   dauv.c AUV_Panel_Code.c AUV_Panel_Code_fn.c ...
HDRS        =   globals.h

all:            dauv

dauv:   $(OBJS)
          cc $(CFLAGS) -align16 -I${INCLUDE}  -o $@  ${OBJS} ${TBX} $(LIBES)

dauv.o:           $(HDRS) $(CODES)

AUV_Panel_Code.o:           AUV_Panel_Code.c AUV_Panel_Code.h
                            $(C) -c -g -w $*.c -G 0

AUV_Panel_Code_fn.o:        AUV_Panel_Code_fn.c AUV_Panel_Code.h
                            $(C) -c -g -w $*.c -G 0
.
.
.
```

Figure 8.13 Modified Makefile for the AUV Simulator

F. Testing and Enhancing the Interface

Nearly always, an initial interface design undergoes extensive testing and some enhancement before it becomes the final version. NPSPD simplifies and speeds the enhancement process. The designer may change panels and actuators directly in the AUV_Panel_Code.c file or using NPSPD as described. Minor changes, such as moving or re-sizing an actuator, can most easily be accomplished by editing the source code files. Major changes, such as introducing a new panel or adding several actuators to an existing

panel, should be accomplished in the NPSPD environment. In either case, the procedures outlined in this chapter must be followed in order to maintain the proper communication between the interface controls and the application.

IX. NPSPD LIMITATIONS AND FUTURE DIRECTIONS

This represents version 1.1 of the NPS Panel Designer and ToolBox. Several important areas of functionality have been designed into the foundational toolbox structures and modules, but have not been fully implemented. This section discusses limitations and suggested future enhancements of NPSPD.

A. Limitations

1. Interactive user specification of actuator detail

All aspects of an actuator should be accessible to the user without having to leave the NPSPD environment. Currently only the parameters common to all actuators (size, label, etc.) can be customized using the Actuator Manager. Future versions of NPSPD should have an editor that is detail-specific, i.e. one that displays different parameters depending on the basic type of the actuator.

2. UNDO key for the last action

All actions, with a few minor exceptions, should be reversible. Major actions that cannot be reversed, such as deleting or clearing a workspace, should prompt the user for confirmation. Currently the NPSPD prompts the user prior to deleting or clearing a workspace. However it does not have an UNDO capability.

3. Complete help

On-line, case sensitive help is essential for even the simplest applications. NPSPD currently provides the user on-line help, but it is not case sensitive and it is general in content. An improved help facility should be developed in future versions.

4. Identify a grouping of actuators

It is frequently necessary to modify two or more actuators in the same way. Examples include moving, sizing, changing color schemes, etc. Currently NPSPD has no

such capability, and incorporating it would greatly simplify the designer's task. Grouping could be temporary or permanent, depending on the purpose.

5. Continued development of basic actuators

This version of NPSPD provides a multitude of basic actuators, but work should be continued to develop more. Additional types might include a text editor incorporated within the fileview actuator, an expanded listview actuator that provides more functionality, etc.

6. Smart Exit/Overwrite

The user should be protected from losing work. This currently can occur if the user exits NPSPD without saving the current workspace(s). It can also occur when workspaces are opened or read from an intermediate file into an existing workspace that has not been saved. Future versions of NPSPD should prompt the user before such actions are completed.

7. Additional actuators partially implemented

Frame, scroll, and cycle actuators have only been partially implemented. Further development work is needed to complete these versatile, but complex controls.

B. Future Directions

This thesis documents many general aspects of user interfaces. It also discusses in detail our attempt to simplify interface design, implementation and testing. The following describes several applicable topics that should be explored in more detail.

1. Efficiency Considerations

The logic of the NPSPD was designed with efficiency as the major goal. Panels and actuators are only drawn when needed due to a change in their appearance, and the time taken to process panels and actuators was minimized as much as possible. The goal was to make the CPU time needed to process and draw the interface panels insignificant as compared to the processing time of the application.

Future work should include benchmark tests for each of the different types of actuators. These tests would provide designers with general guidelines as to the relative

time needed to process and/or draw each of the different actuators. Interfaces could then be designed with these times in mind.

2. NPSPD Design Considerations

The decisions we made in designing the look and feel of the NPSPD were not discussed in detail. Topics such as the default color scheme, the standard size of actuators, the functionality of individual actuators, etc., should be explored in depth. We made design decisions based on our experiences and our research. Future students will undoubtedly bring with them many new experiences and could very well improve upon our work.

3. Portability Considerations

The NPSPD was designed for a specific hardware. An object-oriented programming style was used to modularize the code, but all of the graphics system calls were targeted for the IRIS Window Manager operating system on the SGI workstations. Future work might include looking into porting it to other platforms such as Sun workstations.

X. CONCLUSIONS

The need within the military for effective, flexible and configurable command and control workstations will continue to motivate research into real-time information presentation. The effectiveness of an entire system depends on the user interface's ability to transform data into information and its ability to clearly and simply provide a means to control system operation. Because high quality interface software is costly in time and money, designers will rely on automated development environments that speed and simplify user interface implementation.

The NPS Panel Designer and ToolBox represents the fruit of two man-years of research. Beginning as a tool to assist in the design of a new command and control workstation, it quickly developed into a project of its own. Features and enhancements added during the implementation phase helped to make NPSPD a useful and powerful automated development environment.

Interfaces designed with NPSPD are being used in the development and testing of the Autonomous Underwater Vehicle simulator as well as other simulation projects at the Naval Postgraduate School. Also, NPSPD will be used to support interface design and testing in graphics courses taught at NPS. The applications that are using interfaces developed with the NPSPD have been a tremendous aid in finalizing this version. They have demonstrated many of the capabilities of NPSPD and have identified a few shortcomings.

Students utilizing NPSPD have found it to be easy to use and very flexible. Interface designs can be quickly developed and tested, and then just as quickly fine-tuned. The code generated by the NPSPD is well documented and provides all of the necessary entry points for integration of the interface into an application. The ToolBox functions available to the designer are also well documented and provide a wide range of options for manipulating the interface panels and actuators. Most importantly, processing of the interface panels and actuators is very efficient, saving precious CPU cycles for the target application.

Another strength of the Panel ToolBox is the ease with which additional details may be added to existing actuators or entirely new actuators may be added to the ToolBox. The modular design of the initialization, processing and drawing functions allows easy modification. For example, the single pen stripchart was converted to allow an optional second pen by adding the mode constant for a dual pen mode, the second chart array, and the few lines of code in the stripchart processing and drawing functions.

We hope that the maintenance and improvement of the NPS Panel Designer and ToolBox will continue. If so, it is a tool that can and will be used for a long time. The cost of software development makes NPSPD a valuable tool.

APPENDIX A

NPS PANEL DESIGNER AND TOOLBOX USER'S GUIDE

Introduction and Purpose

The NPS Panel Designer and ToolBox (NPSPD) is an automated development environment that enables design, implementation, modification and testing of customized graphical user interfaces. Pre-designed controls called actuators are provided in the Panel ToolBox and can be placed on a workspace panel, then sized and moved as desired until a final interface layout is developed. The layout can be saved in an intermediate ASCII file for later editing. NPSPD includes automatic generation of compilable source code which can stand alone or be integrated quickly into a developer's application.

Although an object oriented language such as C++ was not used in the design and implementation of the Panel ToolBox, an object oriented approach was used. Distinct abstract data types for the basic actuator and all detailed actuators are defined. Functions are provided to access actuator parameters, details and values.

User

The user is expected to be a systems designer/programmer concerned with application user interfaces. The user should be familiar with the Silicon Graphics Inc. (SGI) IRIS/4D series graphics workstations, operating system and graphics language library as well as the C programming language.

Environment

NPS Panel Designer and ToolBox is designed for use during implementation of application software for SGI IRIS workstations. NPSPD produces compilable C-language source code which can be used in any appropriate application or as a stand-alone program.

User's Guide Organization

Following this introduction, we provide six sections. The Getting Started section discusses in general terms the NPSPD environment and its various components. The Control section describes how to move around the screen and control the functions and operations available in the NPSPD. The Tools section describes in detail how to effectively use each tool and operation available in the design environment. The Modifying Panels and Actuators section discusses the panel and actuator parameters that can be customized. The Source Code Generation and Application Linking section discusses methods of integrating an interface designed with the NPSPD into graphical applications. Finally, the Compilation section presents an example of the instructions required to compile the interface source code produced by NPSPD. In several places in this User's Guide, the reader is referred to the NPSPD Reference Manual for further information.

Suggestions, questions and identification of bugs within the NPS Panel Designer and ToolBox are welcomed. Send such comments to:

Dr. Michael J. Zyda
Naval Postgraduate School
Department of Computer Science
Code CS/Zk
Monterey, CA 93943
zyda@cs.nps.navy.mil

General NPS Panel Designer Usage -- Getting Started

Several files support the operation of NPSPD: npspd, npspd.code and npspd_man.*. The developer should include the Panel ToolBox directory in his working directories path. NPS Panel Designer is started by typing "npspd" at the UNIX system prompt or selecting the npspd executable icon in the IRIS Workspace. The NPSPD copyright notice is displayed while the Palette and workspace are initialized. When NPSPD is ready for the developer's use, the copyright notice will be removed.

The NPSPD environment, shown in Figure A.1, consists of a Palette of actuators and one or more workspace panels. The opening NPSPD copyright panel remains displayed during the initialization sequence, approximately 3 seconds.

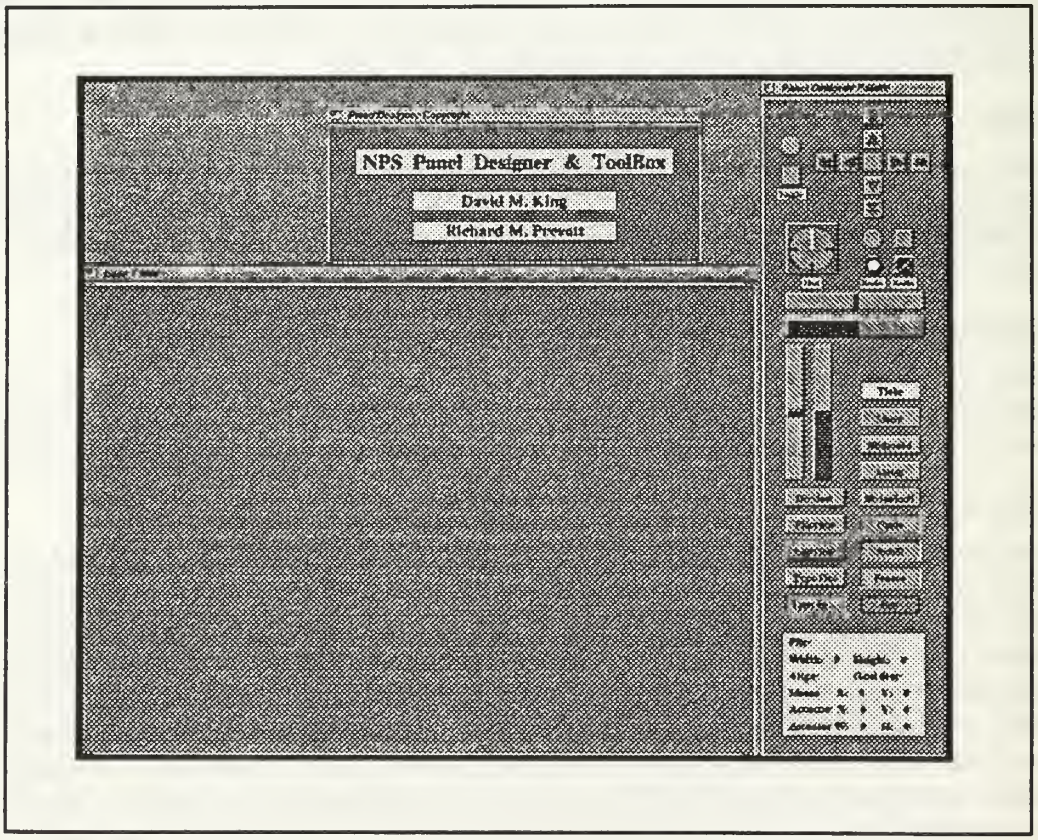

Figure A.1 Opening Layout of NPSPD

Palette and Actuators

The Palette, depicted in Figure A.2, presents all of the actuators provided by the Panel ToolBox for development of user interfaces. The representations for the Buttons, Dials and Sliders are default versions of each of those actuators. All other actuators are made available via labeled selection buttons.

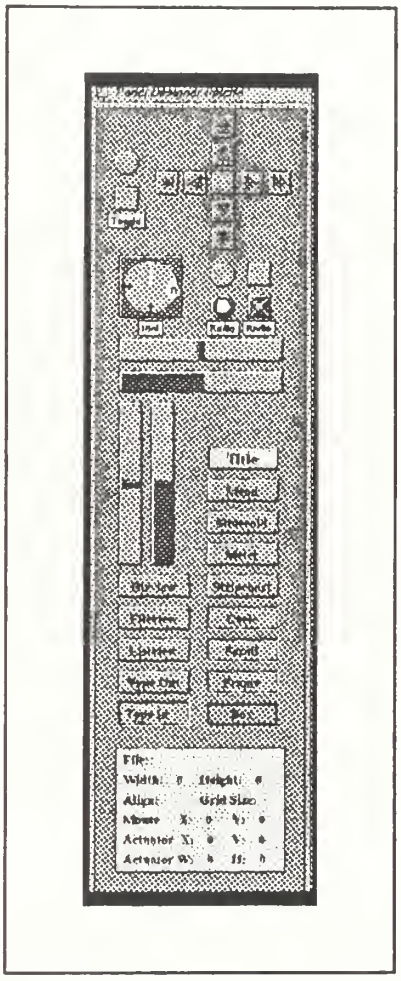

Figure A.2 NPSPD Palette

In the lower portion of the Palette, the Workspace Status Display presents the name and size of the current workspace, the status of workspace auto-alignment and layout grid size, the location in panel coordinates of the mouse cursor, and the location and size of the current actuator on the current workspace. These fields provide continual readouts of layout information useful to the developer.

ToolBox actuators include: momentary buttons, toggle buttons, horizontal and vertical sliders, dials, menus, file-views, list-views and directory-views, custom frames, outline boxes, type-in and type-out fields, meters and stripcharts. Table I presents a complete list

box	meter
button	scroll
cycle	slider
dial	slideroid
dirview	stripchart
fileview	title
frame	typein
listview	typeout
menu	

Table I ToolBox Actuators

of the basic types of ToolBox actuators. The NPSPD Reference Manual provides a detailed description of each actuator.

A basic actuator abstract data-type provides the foundation for all of the diverse ToolBox actuators. Attributes are properties common to all actuators and are recorded in the actuator base structure. Attributes include location and size, value, minimum and maximum values allowed, label, value display format, etc. Each actuator adds unique details to the basic attributes. Details are recorded in a detail structure specific to each different actuator and allow for variation of appearance and function within types of actuators. As an example, the details associated with a Dial include the shape (CIRCLE or RECTANGLE), the number of major and minor tics on the Dial face, and the fine control factor. The NPSPD Reference Manual presents a complete description of actuator attributes.

Workspaces and Panels

Within NPSPD, a workspace is any one of the set of panels onto which the developer positions actuators. It is the blank slate on which the developer designs the user interface. Other panels such as the Palette, Actuator Editor, Color Editor, Panel Editor, etc. are a part of NPSPD but are not available as workspaces.

When NPSPD is initiated, a single workspace panel is presented. Any number of additional workspace panels may be created and modified to participate in the interface under development. All workspaces may be cleared or deleted according to the developers desires. Each workspace panel exactly represents the user interface panel generated by the code generator. Functionality must be included by the application developer.

Control

NPSPD supports three means of interaction control: direct manipulation using the mouse, feature selection using the keyboard and feature selection using pop-up menus. The mouse provides control of interface layout, actuator placement and actuator modification. Functions keys and selected special keys of the keyboard provide the primary means for selection of design tools, editors and managers. Pop-up menus provide an alternate means of selection.

Mouse

The mouse consists of the on-screen cursor and the mouse control unit with its optical sensor, reference pad and three selection buttons. The mouse-cursor is displayed as an arrow in the Palette and as a cross inside all workspace panels. "Left-mouse", "middle-mouse" and "right-mouse" refer to the left, middle and right mouse buttons, respectively, in conjuction with the mouse-cursor position. The location of the mouse determines the current panel and current workspace.

Left-mouse: The left-mouse controls the operation of actuators (e.g., toggle buttons, slide sliders, or set dials). Left-mouse down activates an actuator and its associated host panel, or the panel only if the mouse-cursor is not on an actuator. Left-mouse up de-activates the actuator and/or the associated panel. The left-mouse functions both within NPSPD and within generated user interfaces.

Middle Mouse: The middle-mouse selects an actuator as current within an NPSPD workspace or the Palette. Pressing and releasing the middle-mouse selects an actuator. Pressing and holding the middle-mouse moves or re-sizes an actuator. The middle-mouse functions only within the NPSPD environment and NOT within generated user interfaces.

Right-mouse: The right-mouse controls menu selections. Pressing the right-mouse within any workspace pops up the NPSPD main menu of tools, editors and managers. Positioning and releasing the right-mouse while the desired choice is high-lighted activates NPSPD processing associated with that menu choice. The right-mouse functions both within NPSPD and within generated user interfaces.

Keyboard

NPSPD provides direct access to all of its tools, editors and managers via function keys as described in Table II. Experienced developers speed the development process by use of the function keys rather than the pop-up menu system. NPSPD includes both in keeping with the flexibility requirements of an effective user interface. The insert, delete and backspace keys are active to provide direct actuator copy and delete functions on a workspace. The control key (Ctrl) modifies the behavior of some actuators to yield a fine control operation. Escape provides direct exit from the Panel Designer.

Menu

NPSPD provides alternate access to design tools and features via pop-up menus. Table III presents the NPSPD menu selection hierarchy. Upon pressing the right-mouse button within any workspace, NPSPD presents the main menu. Sub-menus appear as the developer makes a roll-off selection.

Current Workspace and Actuator

NPSPD denotes the workspace on which the mouse-cursor is located as the current panel and the current workspace. Design tool and editor actions take effect in the current workspace. If the mouse-cursor is on the Palette or outside of all of the panels, there is no current panel or current workspace.

Each NPSPD panel may have one actuator selected and designated as the current actuator. Selection via the middle-mouse button displays a white high-light outline around the body of the actuator. NPSPD references the current actuator of the Palette when adding new actuators to a workspace using the middle-mouse button.

F1	On-line Help Manager
F2	Actuator Auto-alignment
F3	Layout Grid Display
F4	Layout Grid Size
F5	Create New Workspace
F6	Clear Current Workspace
F7	Delete Current Workspace
F8	Panel Editor
F9	Actuator Editor
F10	Color Editor
F11	Intermediate File Manager
F12	Source Code Generation Manager
Insert	Copy the current workspace actuator if any
Delete	Delete the current workspace actuator if any
Backspace	Delete the current workspace actuator if any
Ctrl	Fine control of actuator value
Esc	Exit NPS Panel Designer

Table II NPSPD Keyboard Functions

Main Menu Selections:	Sub-menu Selections:
Layout Tools...	
	Auto Align On/Off
	Layout Grid On/Off
	Set Grid Size
Workspace Tools...	
	Create new Workspace
	Clear Current Workspace
	Delete Current Workspace
Panel Editor	
Actuator Editor	
Color Editor	
File Manager	
Code Generation	
Quit	

Table III NPSPD Menu Selections

Workspace Status Display

The lower quarter of the palette, as illustrated in Figure A.2, presents workspace status information concerning the NPS Panel Designer environment.

File: The File field identifies the current workspace by the title associated with it. This is especially useful when multiple workspace panels are in development.

Panel Width and Height: The Panel Width and Height fields present the width and height of the current panel in screen relative units (pixels).

Align: The Align field presents the status ('ON' or 'OFF') of Auto- alignment for the current workspace.

Grid Size: The Grid Size field presents the grid interval spacing in panel relative units.

Mouse X and Y: The Mouse X and Y fields present the X and Y coordinates of the mouse cursor in panel relative units. The reference point (0,0) is the origin of the current panel, the lower left corner.

Actuator X and Y: The Actuator X and Y fields present the X and Y coordinates of the origin of the current actuator in panel relative units. The origin of each actuator is its lower left corner and the reference point for this position is the host panel origin. These fields are useful to position actuators in the same location on separate workspaces or to line up actuators along a common axis within a workspace.

Actuator W and H: The Actuator W and H fields present the width and height of the current actuator in panel relative units.

Tools

The NPSPD provides the user several tools to aid in customizing interface designs. They are broken down into Workspace Tools, Actuator Tools, Environment Tools and Editing and Management Tools. Tools can be invoked by using the specified key or by using the pull down menu.

Workspace Tools

Create New Workspace (F5 key) enables the user to place and size a new workspace panel. Clear Workspace (F6) clears (deletes) any actuators from the current workspace and resets the environment tools to their default values. Delete Workspace (F7) deletes the current workspace. The Clear Workspace and Delete Workspace operations prompt the user to confirm the action before it is completed.

Actuator Tools

Copy Actuator (Insert key) copies the current actuator in the current workspace. The copy is placed to the right of and above the original a distance equal to the original's width and height, respectively. This tool is especially useful for producing copies of customized actuators. It can also be used to add to a group of radio buttons. Delete Actuator (Delete key) deletes the current actuator on the current workspace. Actuators may be moved and resized on a workspace by placing the mouse-cursor on the actuator and holding the left-mouse button down. Figure A.3 maps the selection areas associated with each actuator body to the resulting NPSPD modification.

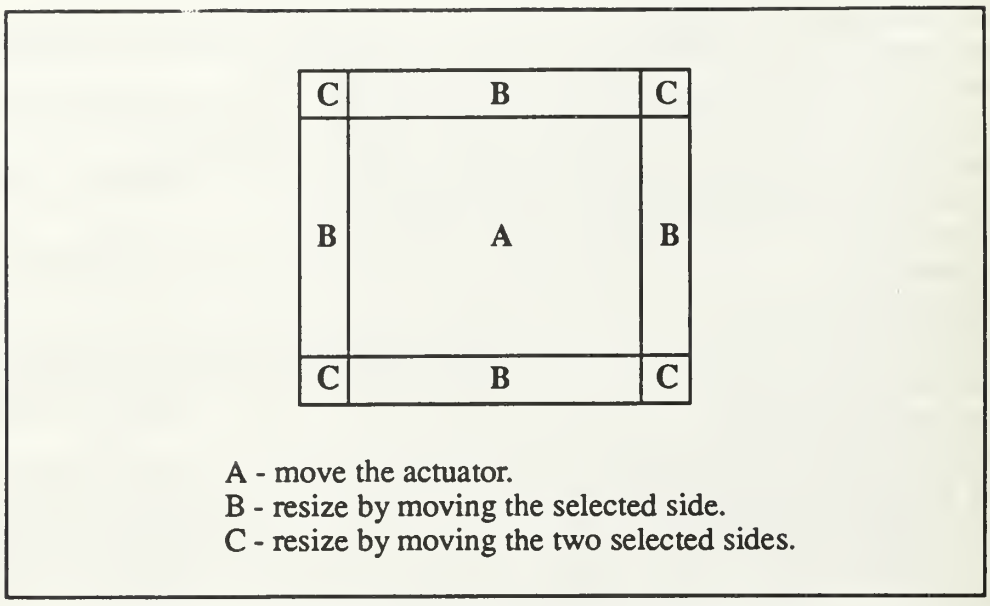

Figure A.3 NPSPD Actuator Move/Resize Areas

Environment Tools

AutoAlign (F2) aligns the reference position of all actuators on the current workspace to the current grid. When this tool is enabled, as indicated in the Workspace Status Display, the reference point of actuators will be moved to the nearest grid intersection corresponding to the current grid size. All subsequent actuators created, moved or resized on the workspace will be aligned similarly. The default state is disabled. Grid Display (F3) displays a grid in the current workspace to help in positioning actuators visually. The default state is off. Grid Size (F4) enables the user to set the size of the grid in the current workspace. The default grid size is 25 pixels. The Grid Size selected for each workspace applies to both AutoAlign and Grid Display. AutoAlign and Grid Display are independent of each other in each workspace, and the environment tools for each workspace are independent of other workspaces.

Editing and Management Tools

The Panel Editor (F8 key) enables the user to interactively modify workspace panels. Figure A.4 is an example of the Panel Manager window. The first typein across the top of the window is used to attach a comment to the panel. This comment will be saved in the intermediate file when the workspace is saved. The second typein is used to change the title of the panel. Changes to this field will be reflected in the title bar of the workspace that is being edited.

The next group of typeins on the left side of the window are used to set the location and size of the panel. Changes to any of these parameters are immediately reflected in the panel. Below the panel location inputs are four typeins that are used to modify the world coordinates of the panel. These values only take effect if the panel is drawn in Screen

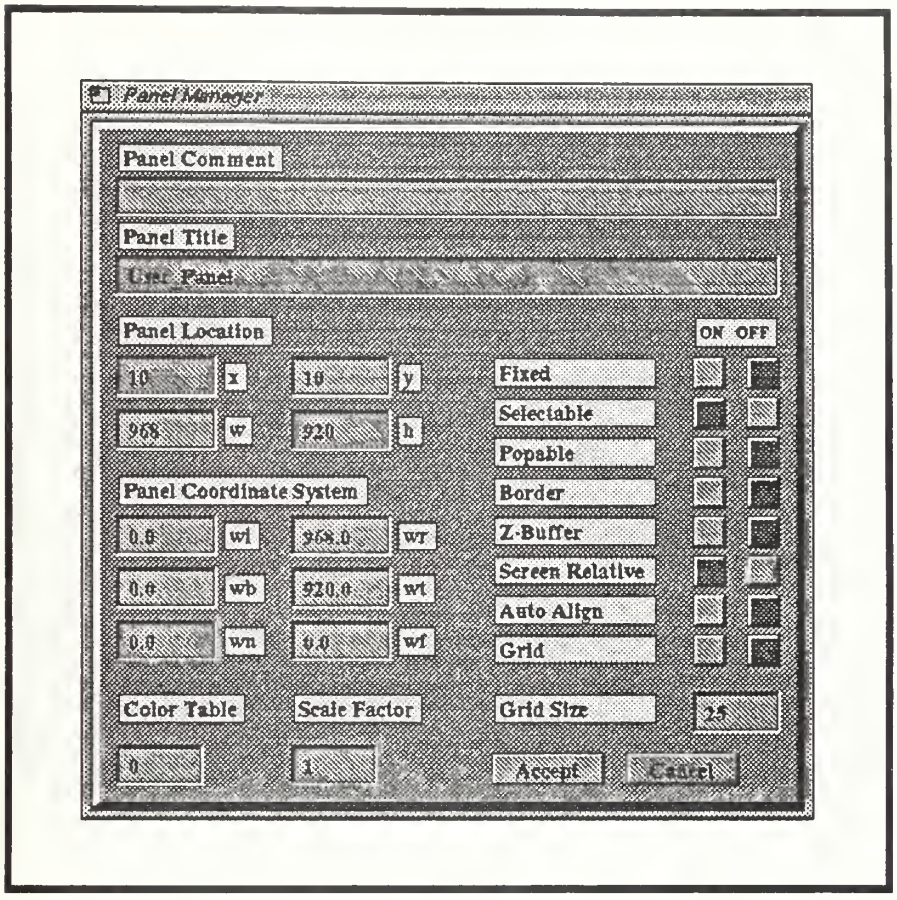

Figure A.4 Panel Editor

Relative mode. Across the bottom of the window are three typeins that enable the user to set the panel's color table, scale factor and grid size.

On the right side of the window are nine sets of radio buttons. These buttons, which can be either ON or OFF, are used to set various flags for the panel. Refer to the User's Manual for a complete explanation of each flag and its meaning.

Finally in the bottom right corner of the window are two buttons. The Accept button is used to make any changes to the panel's parameters permanent. The Cancel button is used to undo any changes made to the panel in the current editing session and restore it to its previous state. Pressing either of these buttons completes the panel editing session and closes the window.

The Actuator Editor (F9 key) enables the user to interactively modify the basic attributes of actuators. Figure A.5 is an example of the Actuator Manager window.

The first typein across the top of the window is the actuator comment field. Comments entered in this typein will be saved in the actuator's permanent comment field in the intermediate file when the actuator's host panel is saved. Below the comment typein is the label typein. This field is used to specify the label for the actuator.

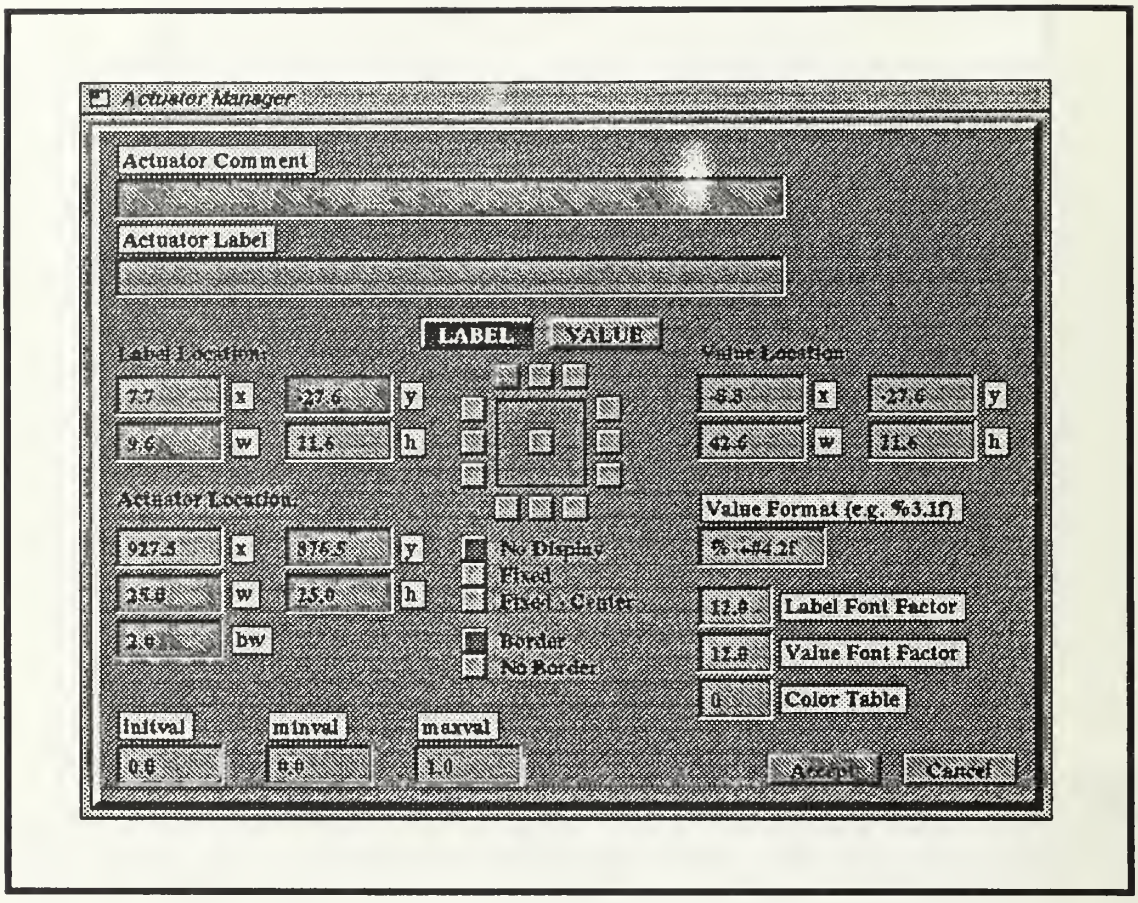

Figure A.5 Actuator Editor

Directly below the label typein are two buttons. The first is marked Label and it is used to control the location of the label string. The second is marked Value and it controls the location of the value output string. The position of these strings is determined by selecting one of the 16 position buttons directly below these two buttons. The 13 relative position buttons surrounding the box are defined as default positions. If a fixed position is desired, either the Fixed button or the Fixed - Center button is selected. The fixed position is then set by entering the appropriate x and y coordinates in either the Label Location typeins or the Value Location typeins.

The actuator's position and size are set with the Actuator Location typeins. The initial, minimum and maximum values associated with the actuator are set with the appropriate typeins in the lower left side of the window.

The format of the value output string is set by entering the appropriate Unix format string in the Value Format typein. The font factor for the label and value strings is set with the Label and Value Font Factor typeins, respectively. Finally, the color table for the actuator is set with the Color Table typein.

The Accept button in the lower right side of Figure A.5 is used to make any modifications to the actuator permanent. The Cancel button is used to undo any changes

made to the actuator in the current editing session and restore it to its previous state. Pressing either of these buttons completes the editing session and closes the window.

The Color Editor (F10 key) enables the user to interactively modify the color of panel backgrounds and individual actuator parts. Figure A.6 is an example of the Color Manager window. The NPSPD allows users to define up to eight custom color tables. Within each

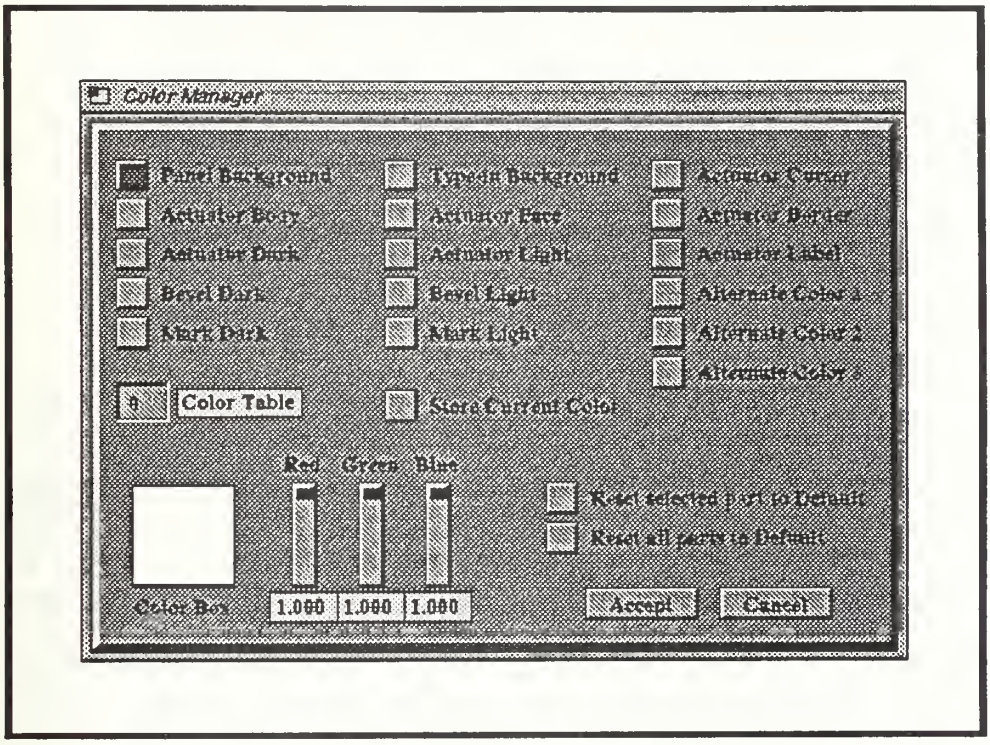

Figure A.6 Color Editor

color table are 24 pre-defined panel and actuator colors. The first eight colors in the table are the basic colors, such as black, white, red, etc. These colors can not be changed by the user. The remaining 16 colors, defined as Panel Background, Actuator Body, etc. can be modified using the Red, Green and Blue sliders. As these sliders are moved, the resulting RGB color is displayed in the Color Box in the lower left corner of the window. The corresponding color in the actuator or panel is also drawn, if applicable. When the desired color is obtained, pressing the Store Current Color button will make the modification permanent. This must be done for each modified color. Colors can be restored to their default values at any time using the two Reset buttons as appropriate. The functionality of the Accept and Cancel buttons is the same as the Actuator and Panel Managers.

The File Manager feature of the NPSPD enables the user to save and recall workspace designs. NPSPD writes all of the pertinent information for a workspace to an ASCII file called the intermediate file. This highly structured file enables the user to store and recall uncompleted work, combine two or more separate designs, and modify designs manually

(outside of the NPSPD environment) by using any text-based editor. Figure A.7 is an example of the File Manager window.

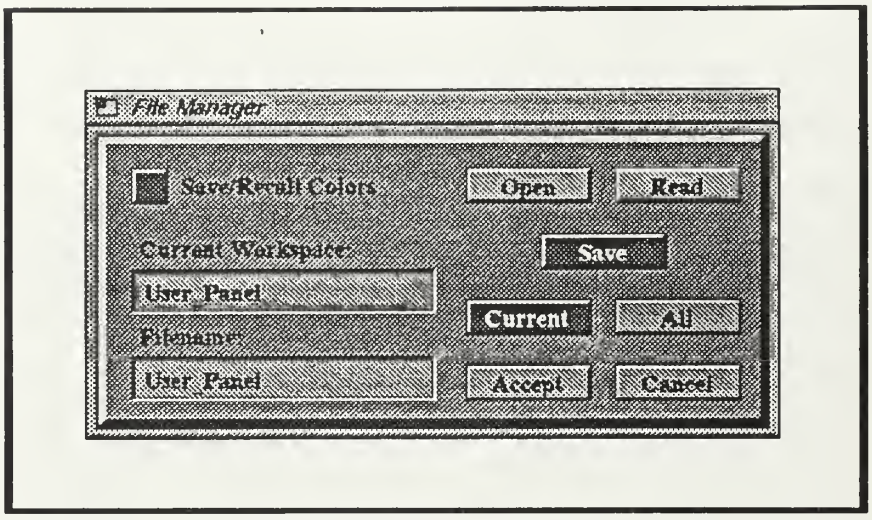

Figure A.7 File Manager

The desired operation (Open, Read or Save) is selected by pressing the appropriate button. Save is the default operation. Open Workspace deletes the Current Workspace and creates all of the panels listed in the named intermediate file. Read Workspace adds the contents of the first panel in the intermediate file to the Current Workspace and creates any subsequent panels listed in the file. Save Workspace saves the contents of the Current Workspace, if Current is selected, or all workspaces to the file specified in the Filename typein. Save/Recall Custom Colors specifies whether to save custom color information to the intermediate file during Save operations, or read custom color information during Open and Read operations. The Current Workspace is specified using the Current Workspace typein. The filename to open, read or save is specified using the Filename typein. Refer to Appendix E for a sample intermediate file.

The Source Code Generation Manager (F12 key) enables the user to generate compilable source code that corresponds to an interface design. Figure A.8 is an example of the Code Manager windowThe Current Workspace typein contains the title of the workspace that was current when the F12 key was pressed. Any valid workspace title can be entered in this field. The default generation mode is for the current workspace only. This can be changed by pressing the appropriate button (either Current or All). If Current is selected, code will be generated for the workspace corresponding to the Current Workspace field. If All is selected, code will be generated for all of the workspaces on the screen regardless of the contents of the Current Workspace field. The name of the output files can be any legal Unix file name, and does not have to be the same as one of the workspaces.

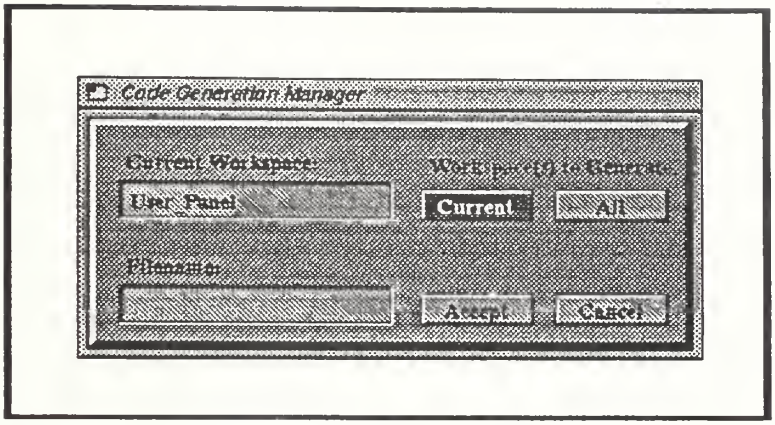

Figure A.8 Code Manager

The Information Manager displays to the user various messages during the NPSPD session. It is opened by the system when an action by the user either causes an error or can not be completed. It is closed by pressing the Continue button. Figure A.9 is an example of the Information Manager window.

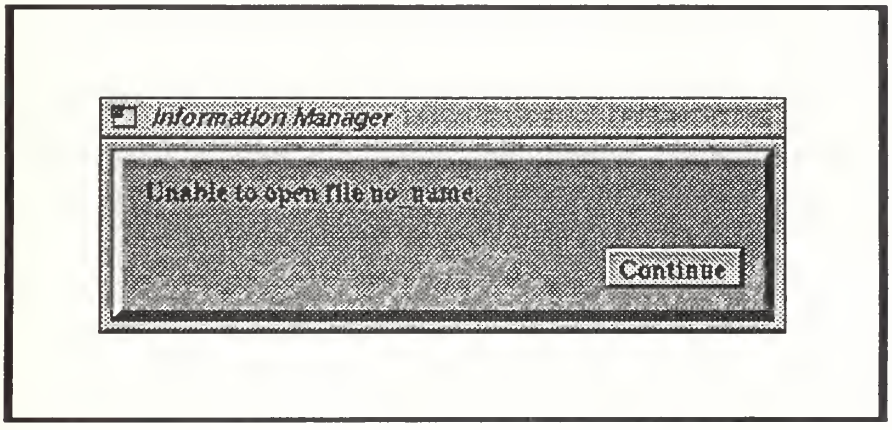

Figure A.9 Information Manager

The Help Manager (F1 key) is an extensive help facility. Figure A.10 is an example of the Help Manager window. The Help Manager contains a Fileview actuator which enables the user to read the on-line manual. The manual is divided into sections as indicated by the index on the first page of section one. The Help Manager panel may be moved to any location on the screen and opened or closed as often as necessary. When the window is closed its contents will be retained so that when it is next opened, the same page will be displayed. The desired set of manual pages is selected by pressing the appropriate button.

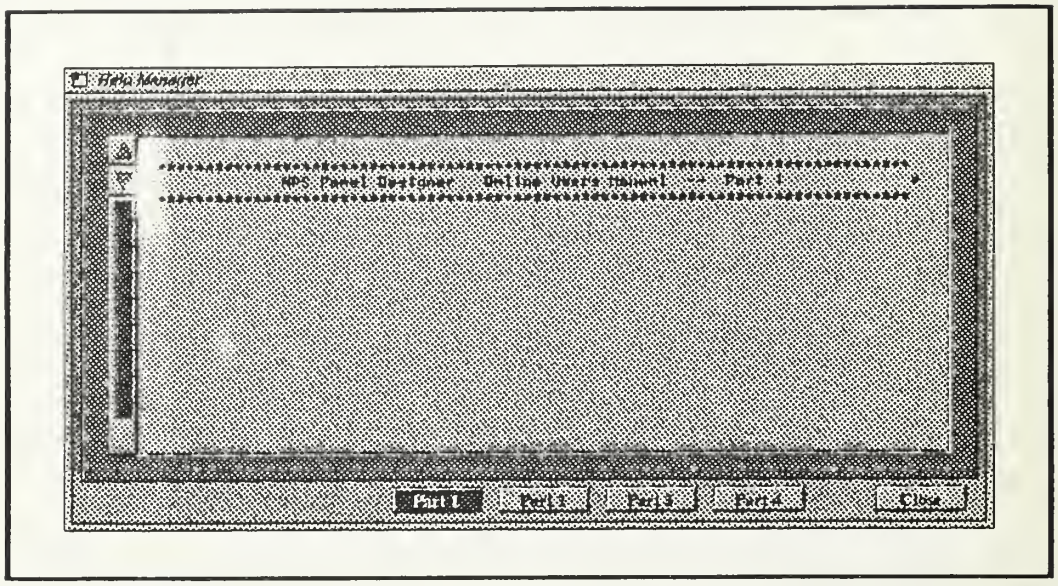

Figure A.10 Help Manager

The user can scroll through the text using either the up and down arrow buttons or the scroll bar on the Fileview. The Help Manager window is closed by either pressing the F1 key or the Close button.

Modifying Panels and Actuators

Most actuator and panel attributes can be modified to produce a custom interface. Basic parameters, such as location, colors, label, etc., can be interactively changed using the tools covered previously in this manual. Detail parameters for actuators must be changed in either the intermediate file or in the generated source code. Refer to the Reference Manual for detailed explanations of this procedure.

Source Code Generation and Application Linking

One of the most powerful features of the NPSPD is its ability to generate source code that corresponds to an interface design. Using the Code Manager, the developer generates source code for the current workspace or all workspaces. The code may then be modified to communicate with the application using clearly defined entry points. The modified code is compiled and linked with the application, providing a custom interface.

There are two methods of integrating an interface designed with the NPSPD into an application. The first method uses the framework of the code generated by the NPSPD and integrates the target application's control features using the entry points provided by the NPSPD. This technique is recommended for users that are designing an application from the beginning.

The second method involves integrating an interface designed with the NPSPD into an existing application by discarding the bulk of the NPSPD code generated for the interface

and using only those functions necessary to initialize, control and draw it. This technique integrates a graphical user interface into applications that either don't have one, or have one that is considered inadequate.

Compilation

Figure A.11 presents an example of the instructions required to compile the interface source code produced by NPSPD. The Panel ToolBox library, *npspanel.a*, must be available to the developer via an appropriate directory path as shown.

```
cc -o user_name User_Panel.c User_Panel_fn.c /nps_path/lib/npspanel.a
                -I/nps_path/include -O2 -align16 -G 0 -lc_s -lgl_s -lfm -lm

    /nps_path is defined as the proper path to the NPS Panel ToolBox library.

    /nps_path = /n/gravy1/work/zyda/npspanel in the current release.

The resulting file 'user_name' may be executed.
```

Figure A.11 NPSPD Source Code Compilation

APPENDIX B

NPS PANEL DESIGNER AND TOOLBOX REFERENCE MANUAL

Introduction

This manual is the reference manual for the data structures and functions of the NPS Panel Designer ToolBox.

Reference Manual Organization

Following this introduction, we provide five sections. The General ToolBox Usage section describes the essential information necessary to integrate the Panel ToolBox into an application. The Panel and Abstract Data-type Definition section describes completely the structure and purpose of the foundational data-types used in the ToolBox. The Actuator Detail Specifications section describes each actuator provided by the ToolBox and the specific access and processing functions related to the actuator. The ToolBox Function Specifications section describes the general access, processing and control functions provided by the ToolBox. Finally, the ToolBox Constants, Global Variables and Support Structures section summarizes the constants, global interface support variables and auxiliary structures provided by the ToolBox.

General ToolBox Usage

The NPSPD ToolBox provides a library of panel and actuator structures with the access and control functions necessary to implement graphical user interfaces. The current C implementation provides no object oriented method of isolating the support data structures from the main program, but it is recommended that the provided access functions be used rather than direct reference to the structures themselves. Both high level and low level creation and management functions are provided for all of the actuators in the ToolBox.

The ToolBox is designed so that default settings for the panels and actuators are sufficient to build a basic interface. Modifications tailor the interface to the needs of the application. Figure B.1 presents an example of the creation and modification of a panel with a single Dial actuator.

```
{
    Panel *p;                          /* Temporary panel pointer          */
    Actuator *a;                       /* Temporary actuator pointer       */

    p = create_panel ();
    set_panel_location(p, 20, 56);
    set_panel_size(p, 720, 534);
    set_attribute(p, visible, TRUE);
    set_attribute(p, fixed, FALSE);
    set_panel_title(p, "User_Panel");
    set_attribute(p, color_table, 1);
    append_panel(p, Panel_List);

    a = create_actuator(dial);
    set_actuator_location(a, 77.5, 119.5);
    set_actuator_size(a, 75, 75, 2);
    set_actuator_label(a, BOTTOM, 10, "Object Rotation Control");
    set_attribute(a, activefunc, rotate_object);
    set_detail(Dial, a, major_tics, 4);
    set_detail(Dial, a, minor_tics, 1);
    set_detail(Dial, a, winds, 1);
    set_detail(Dial, a, finefactor, 0.1);
    insert_actuator(a, p);
}
```

Figure B.1 Creation and Modification Example

ToolBox data structure types and functions listed in the Panel and Abstract Data-type Definition section are available to the programmer within the main routine. Several global variables are made available for management of a panel based system.

A. Initialization Procedures

The Panel ToolBox requires several initialization steps to ensure proper operation. *Initialize_ToolBox()* sets up the ToolBox environment, initializing global state variables, panel management linked lists, the event queue, keyboard buffers, color tables, and fonts. Panel and actuator creations and modifications follow. There is no initialization constraint on either panels or actuators except that the host panel for each actuator must exist before that actuator may be added. Figure B.2 presents the initialization code generated by NPSPD.

```
void initialize_main()                    /* initialize panel environment          */
{

   initialize_ToolBox();                  /* initialize NPS Panel ToolBox          */

   initialize_panels();                   /* Initialize the control panels         */

   initialize_actuators();                /* create the actuators                  */

   initialize_colors();                   /* initialize user defined colors        */

   /*----- initialize all other aspects of main program.                          */

   user_init_queue();                     /* initialize event graphics queue       */
   user_init_menu();                      /* initialize PanelDesigner menus        */
   user_init_cursor();                    /* initialize special cursors            */
   user_init_overlay();                   /* initialize overlay planes & color     */

   /*----- User define initializations are called via user_init_main.             */

    user_init_main();                     /* user defined main initializations     */
}
```

Figure B.2 NPSPD Initialization Sequence

B. Creation Procedures

The Panel ToolBox provides two functions for creation of default panels and default actuators. *Create_panel()*, which requires no arguments, allocates and initializes a panel data structure. *Create_actuator()*, requires an initialization function as its single argument and allocates an actuator basic data structure and unique detail structure as required by the initialization function. Both create functions return a pointer to the new object. Table I presents a list of the initialization functions that may be used as an argument for *create_actuator()*.

C. Insertion Procedures

Once a panel is created, it must be inserted into P*anel_List*, the linked list of panels maintained by the ToolBox. *Insert_panel()* places the new panel at the head of the list. *Append_panel()* places the new panel at the tail of the list. The order of *Panel_List*

basic	dirview	scroll
box	fileview	slider
buffer_act	frame	vbar_slider
button	list_act	vstrip_slider
simple_button	listview	hbar_slider
toggle_button	menu	hstrip_slider
radio_button	arc_meter	slideroid
arrow_button	filled_arc_meter	stripchart
double_arrow_button	dial_meter	dual_stripchart
label_button	filled_dial_meter	hstripchart
cycle	vbar_meter	vstripchart
dial	vstrip_meter	title
square_dial	hbar_meter	typein
round_dial	hstrip_meter	typeout

Table I ToolBox Actuator Initialization Functions

determines the order of panel processing and display. The linked list is traversed from head to tail.

Likewise after an actuator is created, it must be attached to a panel or in some cases to a parent actuator. *Insert_actuator()* and *append_actuator()* add the new actuator to a panel's actuator list, at the head and tail respectively. *Add_sub_actuator()* inserts a specified actuator into another actuator's sub-actuator list (*sa*). Sub-actuators are used by several compound actuators including the Dirview, Fileview and Frame.

D. Modification Procedures

The Panel ToolBox provides a broad compliment of functions for modifying the attributes and details of panels and actuators. The designer directly controls the appearance and function of an interface by way of these modification functions. Modifications may be made both before and after the panel or actuator is added to the interface. *Set_panel_location()* and *set_panel_size()* position and size a panel. *Set_actuator_location()* and *set_actuator_size()* position and size an actuator. *Set_minvalue()* and *set_maxvalue()* set limits on the value range for an actuator. The NPSPD Reference Manual lists and discusses all of the ToolBox functions, their arguments and their use. Two other general modification functions, *set_attribute()* and *set_detail()*, are discussed below.

1. Set_attribute()

Each of the attributes maintained in a panel or an actuator base structure may be modified using the *set_attribute()* function. As depicted in Figure B.1, the arguments for the function call are the panel or actuator pointer, the attribute field name (e.g., visible and activefunc), and the value to be assigned to that attribute. Although some attributes are normally accessed and set by specialized functions such as *set_actuator_size()*, they may also be set using the *set_attribute()* function. An exception applies to the string attributes,

title, *label* and *value_fmt*. These attributes must be set using the specialized functions provided by the ToolBox, *set_panel_title()*, *set_actuator_label()* and *set_value_format()*.

2. Set_detail()

Set_detail() provides the means to modify actuator detail parameters. The function call requires four arguments: the actuator detail data-type, the actuator pointer, the detail field name (e.g., *major_tics* and *minor_tics*), and the value to be assigned to that detail field. A specialized string function, *set_detail_string()*, provides the means to set an actuator detail string field (e.g.,the Typein *buf* field).

3. Binding Modifications

Modifications made to panels and actuators may affect several other aspects of the object. *Fix_panel()* and *fix_actuator()* ensure that all inter-related aspects of the object are adjusted after modifications are completed. Fix functions are specific to each panel and actuator, and they are automatically executed by the ToolBox when any of the insertion functions are called. Normally modifications are made immediately following creation and prior to insertion, thus binding is automatically ensured by the ToolBox. However, modifications may be needed at other points in an application program, possibly in response to user actions. After changes to the attributes of a panel, *fix_panel()* should be explicitly called, and after changes to the attributes or details of an actuator, *fix_actuator()* should be called.

E. Processing Cycle

A graphics application is normally structured with a main program loop that repeatedly calls several functions. These functions typically include input processing, followed by interface display update, followed by applications calculations and display update. Figure B.3 presents the main function and processing support functions generated by NPSPD and supported by the Panel ToolBox.

Control of the interface consists of processing the mouse, keyboard and other device inputs in *process_program_queue()* and processing the interface panels and actuators based on those inputs in *process_panels()*. The ToolBox manages the necessary state variables for mouse position, button action and keyboard action. *Reset_ToolBox_Q()* and *process_ToolBox_Q()* manage the event queue with respect to the interface. Event tokens are also passed to the application program via *user_process_queue()*. *Process_panels()* manages the selected panel and selected actuator ensuring that actuator state and value reflect the user mouse and keyboard inputs.

F. Processing Techniques

The Panel ToolBox supports optional, developer defined action functions that are executed during user activation of a panel and/or actuator. Panels and actuators have three pointers that may be set to reference the developer defined functions. These three attributes are *downfunc*, *activefunc* and *upfunc*. If they are assigned application functions, *downfunc* executes once when the left-mouse button transitions down, *upfunc* executes once when the left-mouse button transitions up, and *activefunc* executes each time *process_panels()* is

```
main()
{

    initialize_main();                  /* initialize main program          */

    forever {                           /* Panel main loop                  */

        control_program(Panel_List);    /* process controls and queue       */
        draw_control_panels(Panel_List); /* draw user control panels & acts */
        user_display();                 /* handle to call user functions    */
    }
}

void control_program                    /* Control program operation        */
(
    PanelList *panel_list               /* specified panel list             */
)
{
    process_program_queue();            /* Process the graphics event queue */
    process_panels(panel_list);         /* Control panels based on user input */
}

void process_program_queue ()           /* Process graphics event queue     */
{
    short TOKdevice,                    /* Graphics event queue device token */
          TOKvalue;                     /* Graphics event queue token value */

    reset_ToolBox_Q();                  /* Prepare ToolBox for input process */

    while ( qtest() ) {                 /* Process all tokens available     */

        TOKdevice = qread(&TOKvalue);
                                        /* Standard ToolBox input processing */
        process_ToolBox_Q(Panel_List, TOKdevice, TOKvalue);

        switch( TOKdevice ) {           /* User Program specific Q processing */

        case RIGHTMOUSE:                /* Right Mouse Controls Menus       */
         if ( TOKvalue == DOWN )        /* on TransitionDown process menu   */
         user_process_menu();           /* User defined menu processor      */
         break;

        } /* end switch */

    /*----- User defined queue function receives all TOKENs processed.      */

    user_process_queue(TOKdevice, TOKvalue);

    } /* end while qtest() */
}
```

Figure B.3 NPSPD Processing Functions

called in the application main loop. These function references provide a powerful control link for the interface developer.

The state of each panel and the state and value of each actuator is available to the application program. State testing functions including *is_active()*, *is_visible()*, *is_selectable()* and *test_flag()* return a Boolean result. State flags may be altered under application control using the *set_attribute()* function or the more specific *set_flag()* and *clear_flag()* functions. *Set_value()* and *get_value()* modify and access an actuator's value.

Special effects may be produced by selectively controlling a panel's or actuator's state, particularly the *visible* and *selectable* flags. *Set_panel_invisible()* and *set_panel_visible()* provide the means to build an effective multi-panel interface.

G. Display Considerations

The Panel ToolBox manages the display of all interface panels and actuators. Drawing occurs only when a change of state or value necessitates an update of the appearance. Actuators are drawn in the reverse order of the host panel's actuator linked list. Thus if two actuators overlap, the one inserted closest to the head of the panel's list is drawn on top.

The ToolBox provides eight modifiable color tables to support multi-color interface designs. Each panel and actuator references one of the color tables as specified by the *color_table* attribute. Changing the *color_table* index or directly modifying the color tables using *define_color_table()* under application control can produce useful effects in the interface.

H. Efficiency Considerations

The Panel ToolBox optimizes processing and drawing algorithms so as not to degrade real-time applications. Panels and actuators each have their own set of specific variables that allow them to be customized for a particular use. For example, panels can be designed so that they are only visible when they are needed, saving screen space and CPU cycles. Similarly actuators can be designed so that they are not selectable, effectively making them output devices (e.g., a Dial, which is normally an input device, can be configured to display the output of a function or operation).

A panel is re-drawn completely only when required by a move or re-size action. During other processing and drawing cases, only those actuators which have been altered and those which have been specifically designated for redraw are drawn. The ToolBox determines visibility and need for redraw at high levels within its hierarchical program flow and prevents excess low level processing when it is not required. The ToolBox processes only the selected panel, if any. While processing panels, the selected actuator on each panel, if any, and the actuators requiring automatic processing are processed.

Panel and Actuator Abstract Data-type Definition

Panel Abstract Data-Type Definition

The NPS Panel ToolBox provides two foundational abstract data types, Panel and Actuator. This section specifies the attributes of a Panel object.

```
typedef struct panel_type {

    long        id;                     /* Unique panel identifier              */
    long        gid;                    /* MEX window identifier for panel      */
    long        redraw_cnt;             /* Count of required redraws            */
    long        act_redraw;             /* True if any actuator needs redraw    */
    char        *comment;               /* Description of panel                 */

    Boolean     visible;                /* Panel visible?                       */
    Boolean     selectable;             /* Panel selectable?                    */
    Boolean     popable;                /* Panel popable when moused?           */
    Boolean     active;                 /* Panel active or not?                 */
    Boolean     fixed;                  /* Panel fixed or variable size?        */
    Boolean     border;                 /* Include border?                      */
    Boolean     screen_relative;        /* World coords are screen relative?    */
    Boolean     zbuffer;                /* Z-buffer is on?                      */
    Boolean     autoalign;              /* Auto alignment to grid?              */
    Boolean     griddraw;               /* Panel grid displayed?                */


    long        x, y, w, h;             /* Origin of panel, width and height    */
    Coord       wl, wr, wb, wt, wn, wf; /* World coordinate system              */
    Object      vobj;                   /* Object holding world coord trans     */
    float       ppu;                    /* Pixels per unit of world distance    */
    float       scale_factor;           /* Scale factor for all actuators       */
    Coord       gridsize;               /* Grid size for this panel             */

    char        title[MAX_STR_LEN+1];   /* Panel title                          */
    Link_list   *keyboard_buffer;       /* Pointer to panel's keyboard buffer   */
    long        color_table;            /* Index of color table for panel       */

                                        /* Function reference pointers          */
    void        (*initfunc)(struct panel_type*);
    void        (*delfunc)(struct panel_type*, struct panel_list_type*);
    void        (*fixfunc)(struct panel_type*);
    void        (*downfunc)();
```

```
    void        (*activefunc)();
    void        (*upfunc)();
    void        (*drawfunc)(struct panel_type*);
    void        (*bkgndfunc)(struct panel_type*);
    void        (*dumpfunc)(struct panel_type*);

    struct actuator_type
                *al_head,           /* Actuator list head                  */
                *al_tail,           /* Actuator list tail                  */
                *ca;                /* Current actuator for panel          */
    struct list_node_type
                *al_auto;           /* List of acts with auto processing   */
    struct panel_type
                *prior,             /* Pointer to prior panel              */
                *next;              /* Pointer to next panel               */
} Panel;
```

Actuator Abstract Data-Type Definition

```
typedef struct actuator_type {

    long      id;                       /* Unique actuator identifier          */
    long      group_id;                 /* Group identifier number             */
    long      type;                     /* Actuator type                       */

    long      redraw_cnt;               /* Count of required redraws           */
    Device    key;                      /* keyboard equivalent if any          */
    char      *comment;                 /* Description of actuator             */

    Boolean   active;                   /* Actuator active or not?             */
    Boolean   visible;                  /* Actuator displayed or not?          */
    Boolean   selectable;               /* Actuator selectable or not?         */

    Coord     x, y, w, h;               /* Location, size in world coords      */
    float     bw;                       /* Bevel width in world coords         */
    float     scale_factor;             /* Scale factor for sub-actuators      */
    long      color_table;              /* Index of color table for actuator   */

    char      label[MAX_STR_LEN+1];     /* Actuator label                      */
    float     label_font_factor;        /* Scale factor for label font         */
    Coord     lx, ly, lw, lh, lbx, lby; /* Label loc, width, height & border   */
    long      l_location;               /* Relative location for label disp.   */

    float     val, initval;             /* Current value and reset value       */
    float     minval, maxval;           /* Minimum and maximum value for act   */
    char      value_fmt[MAX_FMT_LEN+1];/* Format string for value display     */
    float     value_font_factor;        /* Scale factor for value font         */
    Coord     vx, vy, vw, vh, vbx, vby; /* Value loc, width, height & border   */
    long      v_location;               /* Relative location for value disp.   */

                                        /* Function reference pointers         */
    void      (*initfunc)(struct actuator_type*);
    void      (*addfunc)(struct actuator_type*,struct panel_type*);
    void      (*addsubfunc)(struct actuator_type*,struct actuator_type*);
    void      (*delfunc)(struct actuator_type*);
    void      (*fixfunc)(struct actuator_type*);
    Boolean   (*pickfunc)(struct actuator_type*,Coord,Coord);
    void      (*newvalfunc)(struct actuator_type*,Coord,Coord);
    void      (*processfunc)(struct actuator_type*);
    void      (*downfunc)(struct actuator_type*);
    void      (*activefunc)(struct actuator_type*);
    void      (*upfunc)(struct actuator_type*);
    void      (*drawfunc)(struct actuator_type*);
    void      (*dumpfunc)(struct actuator_type*);
```

```
    Void        *detail;                /* Pointer to actuator detail data      */
    long        detail_size;            /* Size of actuator detail struct.      */

    Void        *user_data;             /* Pointer to User specific data.       */

    Panel       *panel;                 /* Pointer to host panel                */

    struct actuator_type
                *pa,                    /* Pointer to host {parent} actuator    */
                *prior,                 /* Pointer to prior act of host list    */
                *next,                  /* Pointer to next actuator of panel    */
                *ga,                    /* Pointer to group-actuator ring       */
                *sa,                    /* Pointer to sub-actuator list         */
                *ca;                    /* Pointer to current sub-actuator      */
} Actuator;
```

Panel and Actuator Attribute Field Definitions:

For each attribute field name, the following information is provided:
field name followed by attribute data type {in braces},
abstract data type to which the attribute field belongs,
attribute definition and use, and
additional references.

act_redraw {long}

Panel field:

Act_redraw records whether or not any of the Panel's Actuators needs to be redrawn to bring the display up to date in both the front and back buffers. It is set at the same time an Actuator's redraw_cnt is set by set_redraw(). It is cleared when all of the Actuators are displayed correctly. This field permits the Panel ToolBox to efficiently minimize overhead when no Actuator needs to be drawn.

see also:
redraw_cnt, set_redraw().

active {Boolean}

Actuator field:

Active is TRUE when the Actuator is selected with the left mouse button or a key equivalent and for as long as the left mouse button or key equivalent is down. When the button is release, active is reset to FALSE. Newvalfunc uses this state variable to properly determine Actuator value or to reset it to its non-selected appearance.

Panel field:

Active is TRUE when the left mouse button is pressed while inside the boundary of the Panel. An Actuator need not be selected.

see also:
key_equivalent, newvalfunc.

activefunc {function pointer}

Actuator field:

Activefunc is one of three Actuator action function pointers. If defined by the User, activefunc is called once each main processing cycle as long as the selected Actuator is active. It is supplied with a pointer to the selected Actuator when it is called.

Panel field:

Activefunc is one of three Panel action function pointers. If defined by the User, activefunc is called once each main processing cycle as long as the selected Panel is active. It is supplied with a pointer to the selected Panel when it is called.

see also:
downfunc, upfunc.

addfunc {function pointer}

Actuator field:

Addfunc is one of five Actuator modification function pointers. If defined for an Actuator, addfunc is called to provide specialized initialization during the process of adding the Actuator to a Panel.

see also:

addsubfunc, delfunc, fixfunc, initfunc, "Modification Functions".

addsubfunc {function pointer}

Actuator field:

Addsubfunc is one of five Actuator modification function pointers. If defined for an Actuator, addsubfunc provides specialized initialization during the addition of a sub-actuator to its parent. It is called by add_sub_actuator() after basic initialization of the sub-actuator.

see also:

addfunc, add_sub_actuator(), delfunc, fixfunc, initfunc, "Modification Functions".

al_auto {List_node*}

Panel field:

Al_auto is the head pointer of the list of Actuators that require automatic processing during each cycle of process_panels(). An Actuator is added to al_auto by fix_actuator() if the Actuator has a defined processfunc.

see also:

fix_actuator(), processfunc.

al_head {Actuator*}

Panel field:

Al_head is the head pointer of the doubly linked list of Actuators that belong to the Panel. An Actuator is added to the head of the list by insert_actuator() or to the tail of the list by append_actuator().

see also:

al_tail, append_actuator(), insert_actuator(), next, prior.

al_tail {Actuator*}

Panel field:

Al_tail is the tail pointer of the doubly linked list of Actuators that belong to the Panel. An Actuator is added to the head of the list by insert_actuator() or to the tail of the list by append_actuator().

see also:

al_head, append_actuator(), insert_actuator(), next, prior.

autoalign {Boolean}

Panel field:

Autoalign is a special Panel field used by the Panel Designer to control auto alignment of Actuators to the specified grid interval for that Panel. Autoalign is TRUE when Actuators are to be aligned to the Panel Designer grid.

see also:

griddraw, gridsize.

bkgndfunc {function pointer}

Panel field:

Bkgndfunc is one of two Panel display function pointers. If defined for a Panel, bkgndfunc provides a User designed background for the Panel. It is called by draw_panel() after the background is cleared when the Panel has been marked for redraw.

see also:

drawfunc.

border {Boolean}

Panel field:

Border controls whether the Panel is created with or without an IRIS window manager border. Default setting is TRUE.

bw {Coord}

Actuator field:

Bw is the Actuator bevel width in Panel coordinates. Positive bw will cause the Actuator to appear raised, negative bw will cause it to appear recessed, and zero bw will produce no bevel. The displayed and pickable dimensions of the Actuator are increased by the absolute value of bw.

see also:

pickfunc, PICKACT(), h, x, y, w.

ca {Actuator*}

Actuator field:

Ca is a reference pointer to an Actuator's current Actuator. Maintenance and use of the ca field is the responsibility of each Actuator. It usually provides a reference to the sub-actuator being operated with the mouse.

Panel field:

Ca is a reference pointer to a Panel's current Actuator. It provides a reference to the Actuator being operated with the mouse.

color_table {long}

Actuator field:

Color_table indicates which of the global color tables is to be used to draw the Actuator. The default color table is 0.

Panel field:

Color_table indicates which of the global color tables is to be used to draw the Panel. The default color table is 0.

see also:

set_actuator_color(), set_panel_color().

comment {char*}

Actuator field:

Comment is a special reference used by the Panel Designer to allow an optional, User specified comment to be associated with each Actuator in the intermediate file when it is saved.

Panel field:

Comment is a special reference used by the Panel Designer to allow an optional, User specified comment to be associated with each Panel in the intermediate file when it is saved.

delfunc {function pointer}

Actuator field:

Delfunc is one of five Actuator modification function pointers. If defined for an Actuator, delfunc is called by delete_actuator() to provide specialized data structure deletion during the process of deleting the Actuator from a Panel. It is called prior to deletion of the detail and basic data structures.

Panel field:

Delfunc is one of three Panel modification function pointers. If defined for a Panel, delfunc is called by delete_panel() to provide specialized data structure deletion during the process of deleting the Panel from a list of Panels. It is called after deletion of all of the Actuators associated with the Panel.

see also:

addfunc, addsubfunc, fixfunc, initfunc, "Modification Functions", delete_actuator(), delete_panel().

detail {Void*}

Actuator field:

Detail provides a pointer to the Actuator's specific detail data structure. An Actuator allocates memory and assigns values to the detail parameters during execution of its initfunc. Each Actuator inherits all of the attributes of the base

class Actuator and adds specific details, if any. Values within the detail structure may be referenced using the ACCESS() macro, the set_detail() macro, or by declaring an auxiliary detail pointer.

example:
(three methods to set the same detail parameter)

```
Actuator *a = create_actuator(button);
Button *ad = (Button*)a->detail;

ACCESS(Button, a, shape) = RECTANGLE;
set_detail(Button, a, shape, RECTANGLE);
ad->shape = RECTANGLE;
```

see also:
ACCESS(), detail_size, initfunc, set_detail().

detail_size {long}

Actuator field:
Detail_size is the size in bytes of the Actuator-specific detail data structure. It is set by initfunc and may be used to determine the amount of data to transfer when copying an Actuator.

see also:
detail, initfunc.

downfunc {function pointer}

Actuator field:
Downfunc is one of three Actuator action function pointers. If defined by the User, downfunc is called once when the left mouse button transitions down selecting the Actuator. It is supplied with a pointer to the selected Actuator when it is called.

Panel field:
Downfunc is one of three Panel action function pointers. If defined by the User, downfunc is called once when the left mouse button transitions down selecting the Panel. It is supplied with a pointer to the selected Panel when it is called.

see also:
activefunc, upfunc.

drawfunc {function pointer}

Actuator field:
Drawfunc is a pointer to the Actuator's drawing routine which renders the graphical representation of the Actuator in the IRIS bitplanes. It is initialized during execution of initfunc and it is called by draw_actuator().

Panel field:
Drawfunc is one of two Panel display function pointers. If defined for a Panel, drawfunc provides a User designed drawing algorithm for the Panel. It is called

by draw_panel() if the Panel is visible and if redraw is required or an actuator has changed appearance. If drawfunc is not defined, a default drawing algorithm is used which calls the drawfuncs for each Actuator.

see also:
draw_actuator(), draw_panel(), initfunc.

dumpfunc {function pointer}

Actuator field:
Dumpfunc is a pointer to an Actuator specific function that is called to dump the values of the detail data structure. If defined, it is called in conjunction with the generalize dump_actuator() function when saving all Actuator parameters to a User specified file. [Note. None of the Actuator dumpfuncs have been defined in this release.]

Panel field:
Dumpfunc is a pointer to a User defined, specialized function that is called to dump the values of the user_data structure. If defined, it is called in conjunction with the generalized dump_panel() function when saving all Panel parameters to a User specified file.

fixed {Boolean}

Panel field:
Fixed controls whether or not the Panel is fixed (TRUE) or variable sized (FALSE). Default setting is FALSE.

fixfunc {function pointer}

Actuator field:
Fixfunc is one of five Actuator modification function pointers. If defined for an Actuator, fixfunc is called by fix_actuator() to provide specialized correction to the Actuator's size and appearance after changes have been made to any of its parameters.

Panel field:
Fixfunc is one of three Panel modification function pointers. If defined for a Panel, fixfunc is called by fix_panel() to provide specialized correction to the Panel after changes have been made to any of its parameters or to Actuator locations or sizes.

see also:
addfunc, addsubfunc, delfunc, fix_actuator(), fix_panel(), initfunc, "Modification Functions".

ga {Actuator*}

Actuator field:
Ga is a pointer used to implement a ring of Actuators which are associated within a group. Actuators are grouped to allow them to modify one another's value when any of them are active. (e.g. Radio buttons use the group ring to unset any other 'on' button within the group when one is selected. An Actuator is added to a group

on a Panel based on its group_id by either add_actuator_to_group() or reset_groups(). It may be removed from its associated group using remove_from_group().

see also:
add_actuator_to_group(), group_id, remove_from_group(), reset_groups().

gid {long}

Panel field:
Gid is the IRIS window manager graphics id for the window within which the Panel is drawn.

griddraw {Boolean}

Panel field:
Griddraw is a special Panel field used by the Panel Designer to control drawing of the alignment grid in the background of the Panel. When Griddraw is TRUE the alignment grid is drawn as part of the Panel background.

see also:
autoalign, gridsize.

gridsize {long}

Panel field:
Gridsize is a special Panel field used by the Panel Designer to control the size of the alignment grid for the Panel. Gridsize may be set to any value. Panel Designer provides gird size menu selections of 5, 10, 25, 50, 75 and 100 units.

see also:
autoalign, griddraw.

group_id {long}

Actuator field:
Group_id is used to associate Actuators on a Panel within a group ring. Actuators are grouped to allow them to modify one another's value when any of them are active. An Actuator is added to a group with matching group_ids by either add_actuator_to_group() or reset_groups(). It may be removed from its associated group using remove_from_group().

see also:
add_actuator_to_group(), ga, remove_from_group(), reset_groups().

h {Coord in Actuator, long in Panel field}

Actuator field:

H is the height of the Actuator in Panel relative coordinates. Bevel width (bw) is outside of height. H may be changed at any time to alter the displayed height of the Actuator. Fix_actuator() should be called after such a change.

Panel field:

H is the height of the Panel in screen relative units (pixels). H does not include pixels used to draw the IRIS window manager border if one is included. H may be changed at any time to alter the displayed height of the Panel. Fix_panel() should be called after such a change.

see also:

x, y, w, bw, fix_actuator(), fix_panel().

id {long}

Actuator field:

Id is a unique identification number provided by the Panel ToolBox to each Actuator at the time of its creation. Default id's are negative so that the User may a use different meaningful constants if desired.

Panel field:

Id is a unique identification number provided by the Panel ToolBox to each Panel at the time of its creation. Default id's are negative so that the User may a use different meaningful constants if desired.

see also:

create_actuator(), create_panel().

initfunc {function pointer}

Actuator field:

Initfunc is one of five Actuator modification function pointers. It is called by create_actuator() to provide initialization specific to each type of Actuator. Initfunc is passed to create_actuator() as a parameter to facilitate easy addition of new types of Actuators to the Panel ToolBox.

Panel field:

Initfunc is one of three Panel modification function pointers. If defined, it is called by fix_panel() to provide User designed initialization related to that Panel.

see also:

addfunc, addsubfunc, create_actuator(), delfunc, fix_panel(), fixfunc, "Modification Functions".

initval {float}

Actuator field:

Initval provides an optional, User specified initial value for each Actuator. It is used by Actuator fixfuncs to reset the val field. Default value, which depends on the particular Actuator, is usually the minval.

see also:

fixfunc, minval, maxval, val.

key {long}

Actuator field:

Key is an optional, User specified identifier for a device to be used as an Actuator key equivalent. If key is defined, Insert_actuator() and append_actuator() queue the appropriate device. Pressing the key is the same as mouse activation with the left mouse button.

see also append_actuator(), insert_actuator().

keyboard_buffer {KeyList pointer}

Panel field:

Keyboard_buffer is a reference pointer to an optional linked-list character buffer used by the Panel to accept input from the keyboard. If initialized by initialize_keyboard_buffer() and activated by activate_keyboard(), keyboard_buffer will receive all character input from the keyboard that is not explicitly directed by cursor position into an active Typein Actuator. The character input may be processed by test_list() and next_char().

see also:

activate_keyboard(), de_activate_keyboard(), initialize_keyboard_buffer(), next_char(), test_list(), Typein.

l_location {long}

(Actuator field} L_location is a constant which indicates the relative position of the Actuator's label field as depicted in Figure A.1. If l_location is positive, the label is displayed with a background box and if l_location is negative, the label is displayed without a background box. Zero in l_location prevents label display. L_location may take on any of the following values:

LABEL_OFF	0		
BOTTOM_LEFT	1	NB_BOTTOM_LEFT	-1
BOTTOM	2	NB_BOTTOM	-2
BOTTOM_RIGHT	3	NB_BOTTOM_RIGHT	-3
RIGHT_LOWER	4	NB_RIGHT_LOWER	-4
RIGHT	5	NB_RIGHT	-5
RIGHT_UPPER	6	NB_RIGHT_UPPER	-6
TOP_RIGHT	7	NB_TOP_RIGHT	-7
TOP	8	NB_TOP	-8

TOP_LEFT	9	NB_TOP_LEFT	-9
LEFT_UPPER	10	NB_LEFT_UPPER	-10
LEFT	11	NB_LEFT	-11
LEFT_LOWER	12	NB_LEFT_LOWER	-12
CENTER	13	NB_CENTER	-13
FIXED	14	NB_FIXED	-14
FIXED_CENTER	15	NB_FIXED_CENTER	-15

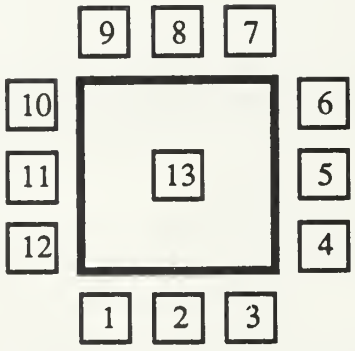

Figure B.1 Label Locations

FIXED and NB_FIXED are special constants that allow the User to explicitly specify label position using lx, ly, lw, lh, lbx, and lby. The ToolBox will only adjust lw and lh so that the background box will fully include the label. FIXED_CENTER and NB_FIXED_CENTER are similar to FIXED with the added property that the text of the label will be centered within the User specified label area.

see also:
compute_label_location(), compute_location(), set_actuator_label(), set_label_location(), set_label_size(), lx, ly, lw, lh, lbx, lby.

label {string}

Actuator field:
Label is an optional, User specified string which appears near the Actuator in a location specified by l_location.

see also:
l_location, set_actuator_label(), set_label_location(), set_label_size(), lx, ly, lw, lh, lbx, lby.

label_font_factor {float}

Actuator field:
Label_font_factor specifies the scaling factor which is applied to the Actuator's label font. The scaling is approximately equal to an equivalent point size (e.g.

label_font_factor = 12.0 approximates display of label text in a 12 point font). Any non-negative font_factor is allowed, although reasonably usable font_factors range from 8.0 to 120.0.

lbx {Coord}

Actuator field:

Lbx is the x direction border offset in Panel relative coordinates of the Actuator's label (i.e. the space between the label background box boundary and the text of the label). It is recalculated after changes by fix_actuator().

see also:

lx, ly, lw, lh, lby, fix_actuator().

lby {Coord}

Actuator field:

Lby is the y direction border offset in Panel relative coordinates of the Actuator's label (i.e. the space between the label background box boundary and the text of the label). It is recalculated after changes by fix_actuator().

see also:

lx, ly, lw, lh, lbx, fix_actuator().

lh {Coord}

Actuator field:

Lh is the height in Panel relative coordinates of the Actuator's label, including the background box. It is recalculated after changes by fix_actuator().

see also:

lx, ly, lw, lbx, lby, fix_actuator().

lw {Coord}

Actuator field:

Lw is the width in Panel relative coordinates of the Actuator's label, including the background box. It is recalculated after changes by fix_actuator().

see also:

lx, ly, lh, lbx, lby, fix_actuator().

lx {Coord}

Actuator field:

Lx is the x location in Panel relative coordinates of the Actuator's label. The label is positioned relative to the Actuator's origin (lower left corner).Lx is recalculated after changes by fix_actuator().

see also:

ly, lw, lh, lbx, lby, fix_actuator().

ly {Coord}

Actuator field:

Ly is the y location in Panel relative coordinates of the Actuator's label. The label is positioned relative to the Actuator's origin (lower left corner). Ly is recalculated after changes by fix_actuator().

see also:

lx, lw, lh, lbx, lby, fix_actuator().

maxval {float}

Actuator field:

Maxval is the maximum value that an Actuator may take in its val field. For continuous Actuators, such as Dials or Sliders, maxval is the upper limit on the value of the Actuator. For discrete Actuators, such as Buttons, val is set to maxval when the Actuator is selected or 'ON'.

minval {float}

Actuator field:

Minval is the minimum value that an Actuator may take in its val field. For continuous Actuators, such as Dials or Sliders, minval is the lower limit on the value of the Actuator. For discrete Actuators, such as Buttons, val is set to minval when the Actuator is not selected or 'OFF'.

newvalfunc {function pointer}

Actuator field:

Newvalfunc is one of three Actuator control function pointers. If defined, newvalfunc is called within process_actuator() when an Actuator is first selected with the mouse or key-equivalent, and it is repeated called each processing cycle as long as the Actuator is selected. Newvalfunc computes the Actuator's state and value based on cursor position relative to Panel or Parent-Actuator's origin. When the mouse button or key-equivalent is released, newvalfunc is called a last time to return the Actuator to its in-active state and value.

see also:

pickfunc, processfunc, process_actuator(), val.

next {Actuator* in Actuator, Panel* in Panel field}

Actuator field:

Next provides the forward link in a Panel's list of Actuators (al_head) or a Parent-Actuator's list of Sub-Actuators (sa). When an Actuator is added to or removed

from a Panel or Parent-Actuator, next is appropriately managed by insert_actuator(), append_actuator(), add_sub_actuator(), extract_actuator(), and delete_actuator().

Panel field:

Next provides the forward link in a list of Panels (e.g. Panel_List which is managed by the Panel ToolBox). A Panel is added to a specified list by insert_panel() or append_panel() and removed by delete_panel().

see also:

al_head, al_tail, Panel_List, prior, sa, append_actuator(), insert_actuator(), add_sub_actuator(), extract_actuator(), delete_actuator(), append_panel(), insert_panel(), delete_panel().

pa {Actuator*}

Actuator field:

Pa provides a reference to the Parent-Actuator of each Sub- Actuator. Pa is NULL is the Actuator is not a Sub-Actuator. Pa is managed by add_sub_actuator().

see also:

add_sub_actuator().

panel {Panel*}

Actuator field:

The panel field provides a reference to the host Panel for each Actuator. Sub-Actuators reference the same host Panel as their Parent- Actuator.

see also:

append_actuator(), insert_actuator().

pickfunc {function pointer}

Actuator field:

Pickfunc is an optional one of three Actuator control function pointers. If defined, pickfunc provides the algorithm for determining if the Actuator is selected by the cursor position and left mouse button. If not defined, an efficient default algorithm is used which compares cursor location to Actuator boundary. If the pick algorithm returns TRUE then the Actuator becomes the Selected_Actuator.

see also:

newvalfunc, processfunc, Selected_Actuator

popable {Boolean}

Panel field:

Popable controls whether or not the Panel is "popped" by the Panel ToolBox when it is selected using the cursor and left mouse button. Default setting is FALSE.

ppu {float}

Panel field:

Ppu is the number of pixels per unit dimension in the Panel relative coordinate system. It is used to normalize pixel oriented dimensions (e.g. string widths) in terms of Panel relative coordinates. Ppu is calculated by fix_panel().

see also:

fix_panel()

prior {Actuator* in Actuator, Panel* in Panel field}

Actuator field:

Prior provides the reverse link in a Panel's list of Actuators (al_tail or a Parent-Actuator's list of Sub-Actuators (sa). When an Actuator is added to or removed from a Panel or Parent-Actuator, prior is appropriately managed by insert_actuator(), append_actuator(), add_sub_actuator(), extract_actuator(), and delete_actuator().

Panel field:

Prior provides the reverse link in a list of Panels (e.g. Panel_List which is managed by the Panel ToolBox). A Panel is added to a specified list by insert_panel() or append_panel() and removed by delete_panel().

see also:

al_head, al_tail, Panel_List, next, sa, append_actuator(), insert_actuator(), add_sub_actuator(), extract_actuator(), delete_actuator(), append_panel(), insert_panel(), delete_panel().

redraw_cnt {long}

Actuator field:

Redraw_cnt records the number of times an Actuator must be drawn to bring the display up to date. When an Actuator changes its state or value, redraw_cnt is set to two (2) indicating that both the front and back buffers are incorrect with respect to the Actuator. Redraw_cnt is set using set_redraw(), usually during execution of the Actuator's newvalfunc or processfunc. Set_redraw() also sets the act_redraw field for the host Panel to indicate that at least one Actuator on the Panel must be redrawn. Drawing functions will draw the Actuator only if redraw_cnt is greater than zero, and as the Actuator is drawn in each buffer, redraw_cnt is decremented. Compound Actuators properly set the redraw_cnt for their Sub-Actuators to ensure complete drawing. User code may force a redrawing of any Actuator by using set_redraw().

Panel field:

Redraw_cnt records the number of times a Panel must be redrawn to bring the display up to date after being reshaped or moved or when the background has been disturbed. Redraw_cnt is set to two (2) indicating that both the front and back buffers are incorrect with respect to the Panel and its background. As the Panel is

completely redrawn, redraw_cnt is decremented. If a Panel's redraw_cnt is greater than zero, every Actuator on the Panel is also redrawn. User code may force a redrawing of a Panel by using set_redraw().

see also:
act_redraw, drawfunc, newvalfunc, processfunc, set_redraw().

sa {Actuator*}

Actuator field:
Sa is a reference pointer to an Actuator's optional list of Sub-Actuators. Sub-Actuators are added to the head of an Actuator's sa list by add_sub_actuator() and are linked through their prior and next fields.

see also:
prior, next, addsubfunc, add_sub_actuator()

scale_factor {float}

Actuator field:
Scale_factor is a scaling factor which is applied to all Sub- Actuators of an Actuator when drawn. The default scale_factor is 1.0.

Panel field:
Scale_factor is a scaling factor which is applied to all Actuators of a Panel when drawn. The default scale_factor is 1.0.

screen_relative {Boolean}

Panel field:
Screen_relative controls whether or not the Panel is created with a coordinate system that is screen relative. If screen_relative is TRUE, wl, wb, wn and wf equal 0.0, wr equals w and wt equals h for the Panel. If screen_relative is FALSE, then the Panel coordinate system must be defined by the User. Screen_relative may be changed during execution as long as fix_panel() is called after the change. Default setting is TRUE.

see also:
wl, wr, wb, wt, wn, wf, fix_panel()

selectable {Boolean}

Actuator field:
Selectable controls whether or not the Actuator may be selected and controlled using the mouse cursor and left button. Setting selectable FALSE causes the Actuator to be drawn with a striped overlay. Default setting is TRUE.

Panel field:
Selectable controls whether or not the Panel and its Actuators may be selected and controlled using the mouse cursor and left button. Setting selectable FALSE causes the Panel to be drawn with a striped overlay. Default setting is TRUE.

title {string}

Panel field:

Title is an optional character string which appears in the title bar of the Panel's window if border is set TRUE.

see also:

border

type {long}

Actuator field:

Type indicates what the Actuator is. Type is set by each Actuator's initfunc to one of the following constant values:

BASIC	5
BOX	10
BUFFER_ACT	60
BUTTON	20
CYCLE	30
DIAL	40
DIRVIEW	50
FILEVIEW	70
FRAME	80
LIST_ACT	90
LISTVIEW	190
MENU	100
METER	110
SCROLL	120
SLIDER	130
SLIDEROID	140
STRIPCHART	150
TITLE	160
TYPEIN	170
TYPEOUT	180

see also:

"Actuator Descriptions".

upfunc {function pointer}

Actuator field:

Upfunc is one of three Actuator action function pointers. If defined by the User, downfunc is called once when the left mouse button transitions up de-selecting the Actuator. It is supplied with a pointer to the selected Actuator when it is called.

Panel field:

Upfunc is one of three Panel action function pointers. If defined by the User, upfunc is called once when the left mouse button transitions up de-selecting the Panel. It is supplied with a pointer to the selected Panel when it is called.

see also:

activefunc, downfunc.

user_data {Void*}

Actuator field:

User_data is a reference pointer to an optional, User defined data structure for the Actuator.

v_location {long}

(Actuator field} V_location is a constant which indicates the relative position of the Actuator's value field as depicted in Figure A.2. If v_location is positive, the value is displayed with a background box and if v_location is negative, the value is displayed without a background box. Zero in v_location prevents value display. V_location may take on any of the following values:

LABEL_OFF	0		
BOTTOM_LEFT	1	NB_BOTTOM_LEFT	-1
BOTTOM	2	NB_BOTTOM	-2
BOTTOM_RIGHT	3	NB_BOTTOM_RIGHT	-3
RIGHT_LOWER	4	NB_RIGHT_LOWER	-4
RIGHT	5	NB_RIGHT	-5
RIGHT_UPPER	6	NB_RIGHT_UPPER	-6
TOP_RIGHT	7	NB_TOP_RIGHT	-7
TOP	8	NB_TOP	-8
TOP_LEFT	9	NB_TOP_LEFT	-9
LEFT_UPPER	10	NB_LEFT_UPPER	-10
LEFT	11	NB_LEFT	-11
LEFT_LOWER	12	NB_LEFT_LOWER	-12
CENTER	13	NB_CENTER	-13
FIXED	14	NB_FIXED	-14
FIXED_CENTER	15	NB_FIXED_CENTER	-15

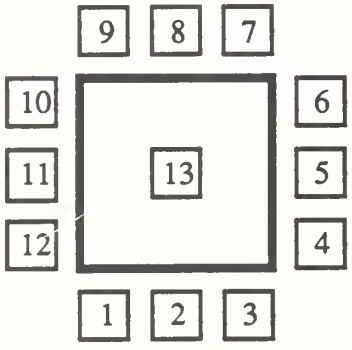

Figure B.2 Value Locations

FIXED and NB_FIXED are special constants that allow the User to explicitly specify value position using vx, vy, vw, vh, vbx, and vby. The ToolBox will only adjust lw and lh so that the background box will fully include the value. FIXED_CENTER and NB_FIXED_CENTER are similar to FIXED with the added property that the text of the value will be centered within the User specified value area.

see also:
compute_value_location(), compute_location(), set_actuator_format(), set_value_location(), set_value_size(), vx, vy, vw, vh, vbx, vby.

val {float}

Actuator field:
Val contains the current value of an Actuator. It is set by the Actuator's newvalfunc or processfunc, or may be set directly by the User. Val is initially set to initval by fix_actuator() and is limited by minval and maxval. For continuous Actuators (e.g. Sliders), val will range any where between minval and maxval, and for discrete Actuators (e.g. Buttons), val is set to maxval when the Actuator is selected or 'ON' and to minval when it is de-selected or 'OFF'. Set_redraw() should be called after val is directly changed by User code.

see also:
initval, maxval, minval, set_redraw(), newvalfunc, processfunc.

value_fmt {string}

Actuator field:
Value_fmt is a string which is use to format the display of an Actuator's value. Format characters are standard as specified by the ANSI C printf function. Value_fmt may be set using set_value_format(). Default value_fmt is "%-+#4.2f" which displays the value left justified with sign, decimal point and 2 digits after the decimal point.

see also:
set_value_format().

value_font_factor {float}

Actuator field:
Value_font_factor specifies the scaling factor which is applied to the Actuator's value font. The scaling is approximately equal to an equivalent point size (e.g. value_font_factor = 12.0 approximates display of value numerals in a 12 point font). Any non-negative font_factor is allowed, although reasonably usable font_factors range from 8.0 to 120.0.

vbx {Coord}

Actuator field:

Vbx is the x direction border offset in Panel relative coordinates of the Actuator's value (i.e. the space between the value background box boundary and the value string). It is recalculated after changes by fix_actuator().

see also:

vx, vy, vw, vh, vby, fix_actuator().

vby {Coord}

Actuator field:

Vby is the y direction border offset in Panel relative coordinates of the Actuator's value (i.e. the space between the value background box boundary and the value string). It is recalculated after changes by fix_actuator().

see also:

vx, vy, vw, vh, vbx, fix_actuator().

vh {Coord}

Actuator field:

Vh is the height in Panel relative coordinates of the Actuator's value display, including the background box. It is recalculated after changes by fix_actuator().

see also:

vx, vy, vw, vbx, vby, fix_actuator().

visible {Boolean}

Actuator field:

Visible controls whether or not the Actuator is drawn on the host Panel. Use fix_actuator() after explicitly changing the visible field. Default setting is TRUE.

Panel field:

Visible controls whether or not the Panel is displayed. Setting visible FALSE or calling set_panel_invisible() will close the Panel's IRIS window making the Panel invisible. Setting visible TRUE or calling set_panel_visible() will create and initialize an IRIS window for the Panel and draw the Panel. Use fix_panel() after explicitly changing the visible field. Default setting is TRUE.

see also:

fix_actuator(), fix_panel(), set_panel_invisible(), set_panel_visible().

vobj {graphics object}

Panel field:

Vobj is a IRIS graphics object containing the viewing transformation in effect for the Panel. It is used by the Panel ToolBox to map a screen relative mouse cursor position into Panel relative coordinates.

vw {Coord}

Actuator field:

Vw is the width in Panel relative coordinates of the Actuator's value display, including the background box. It is recalculated after changes by fix_actuator().

see also:

vx, vy, vh, vbx, vby, fix_actuator().

vx {Coord}

Actuator field:

Vx is the x location in Panel relative coordinates of the Actuator's value display. The value display is positioned relative to the Actuator's origin (lower left corner). Vx is recalculated after changes by fix_actuator().

see also:

vy, vw, vh, vbx, vby, fix_actuator().

vy {Coord}

Actuator field:

Vy is the y location in Panel relative coordinates of the Actuator's value display. The value display is positioned relative to the Actuator's origin (lower left corner). Vy is recalculated after changes by fix_actuator().

see also:

vx, vw, vh, vbx, vby, fix_actuator().

w {Coord in Actuator, long in Panel field}

Actuator field:

W is the width of the Actuator in Panel relative coordinates. Bevel width (bw) is outside of width. W may be changed at any time to alter the displayed width of the Actuator. Fix_actuator() should be called after such a change.

Panel field:

W is the width of the Panel in screen relative units (pixels). W does not include pixels used to draw the IRIS window manager border if one is included. W may be changed at any time to alter the displayed width of the Panel. Fix_panel() should be called after such a change.

see also:

x, y, h, bw, fix_actuator(), fix_panel().

wb {Coord}

Panel field:

Wb specifies the Panel relative coordinate system value for the bottom edge of the Panel (negative y axis). It is set to 0.0 by the Panel ToolBox if screen_relative is TRUE, and it must be set by the User if screen_relative is FALSE.

see also:

wl, wr, wt, wn, wf, screen_relative, fix_panel()

wf {Coord}

Panel field:

Wf specifies the Panel relative coordinate system value for the far extent of the Panel (positive z axis). It is set to 0.0 by the Panel ToolBox if screen_relative is TRUE, and it must be set by the User if screen_relative is FALSE.

see also:

wl, wr, wb, wt, wn, screen_relative, fix_panel()

wl {Coord}

Panel field:

Wl specifies the Panel relative coordinate system value for the left edge of the Panel (negative x axis). It is set to 0.0 by the Panel ToolBox if screen_relative is TRUE, and it must be set by the User if screen_relative is FALSE.

see also:

wr, wb, wt, wn, wf, screen_relative, fix_panel()

wn {Coord}

Panel field:

Wn specifies the Panel relative coordinate system value for the near extent of the Panel (negative z axis). It is set to 0.0 by the Panel ToolBox if screen_relative is TRUE, and it must be set by the User if screen_relative is FALSE.

see also:

wl, wr, wb, wt, wf, screen_relative, fix_panel()

wr {Coord}

Panel field:

Wr specifies the Panel relative coordinate system value for the right edge of the Panel (positive x axis). It is calculated by the Panel ToolBox if screen_relative is TRUE, and it must be set by the User if screen_relative is FALSE.

see also:

wl, wb, wt, wn, wf, screen_relative, fix_panel()

wt {Coord}

Panel field:

Wt specifies the Panel relative coordinate system value for the top edge of the Panel (positive y axis). It is calculated by the Panel ToolBox if screen_relative is TRUE, and it must be set by the User if screen_relative is FALSE.

see also:

wl, wr, wb, wn, wf, screen_relative, fix_panel()

x {Coord in Actuator, long in Panel field}

Actuator field:

X is the x location of the Actuator's lower left corner in Panel relative coordinates within the host Panel or Parent Actuator. X may be changed at any time to alter the displayed location of the Actuator. Fix_actuator() should be called after such a change.

Panel field:

X is the x location of the Panel's lower left corner in screen relative units (pixels). X may be changed at any time to alter the displayed position of the Panel. Fix_panel() should be called after such a change.

see also:

y, w, h, bw, fix_actuator(), fix_panel().

y {Coord in Actuator, long in Panel field}

Actuator field:

Y is the y location of the Actuator's lower left corner in Panel relative coordinates within the host Panel or Parent Actuator. Y may be changed at any time to alter the displayed location of the Actuator. Fix_actuator() should be called after such a change.

Panel field:

Y is the y location of the Panel's lower left corner in screen relative units (pixels). Y may be changed at any time to alter the displayed position of the Panel. Fix_panel() should be called after such a change.

see also:

x, w, h, bw, fix_actuator(), fix_panel().

zbuffer {Boolean}

Panel field:

Zbuffer controls whether or not the Panel is drawn using the IRIS zbuffer and a mode of MSINGLE or MVIEWING. Setting zbuffer to TRUE initializes the zbuffer, and sets the drawing mode to MVIEWING. Setting zbuffer to FALSE sets the drawing mode to MSINGLE. Set_panel() appropriately sets the drawing mode before any Panel is drawn based on this flag. Fix_panel() must be called after a change to zbuffer. Default setting is FALSE.

see also:

fix_panel().

Actuator Detailed Specifications

The NPS Panel ToolBox provides a wide variety of pre-designed user interface objects called Actuators. Each has a distinct function and corresponding appearance. Most Actuators may be selected and controlled using the mouse cursor and left mouse button (referred to as the left-mouse). Several Actuators provide only a display of data and are not directly controlled using the mouse.

Actuators provided in the ToolBox:

box
buffer_act
button
cycle
dial
dirview
fileview
frame
list_act
listview
menu
meter
scroll
slider
slideroid
stripchart
title
typein
typeout

Summary of Actuator Function and Appearance

Box

Box provides a rectangular box with a user defined line width, foreground color and background color. It may be used to visually group related Actuators, bound display areas, divide Panels, etc. A Box background may be color filled or clear.

Buffer_act

Buffer_Act is a sub-actuator which is used in two standard Actuators: Fileview and Typeout. The Buffer_Act takes a character string, or buffer, and displays it using a fixed font size of 12 point and non-proportional spacing. Carriage returns and line feeds are treated as new lines, and the default width is 80 columns.

Button

Button is a basic Actuator which has two states: 'ON' or 'OFF'. The ToolBox provides three functional sub-types of button: simple which is 'ON' only when selected with the left-mouse, toggle which swaps state with each left-mouse selection, and radio which forms into a group with only the most recently selected button 'ON' and the others 'OFF'.

Cycle

Cycle is a compound Actuator which displays one of a set of sub-actuators that have been added to it. The User may advance through the list of sub-actuators by selecting the cycle body outside of the displayed sub-actuator.

Dial

Dial is a basic actuator that provides a means for 360 degree selection and control. Its face can be customized in terms of radius, number of tics and tic size. Its control characteristics can be modified in terms of control mode (wrap or block), winds and fine factor.

Dirview

Dirview is a compound actuator that provides a means to view the contents of a directory and select a file. The current directory is displayed at the top of the Dirview. The accept button copies the currently selected item to the user-defined target if the item is a file name, or changes directories if it is a directory name. The reset button returns the Dirview to the current working directory of the user.

Fileview

Fileview is a compound actuator that provides a means to select and view a file. The default dimensions of the text buffer are 15 lines and 80 columns, with lines exceeding the width of the text buffer automatically wrapped. The width and height of the Fileview actuator can be re-sized. A user specified size limits the buffer size of the displayed file (see Buffer_Act). The accept button is used to copy high-lighted text to the user-defined target.

Frame

Frame is a compound Actuator used to group and display a set of sub-actuators. Actuators added to and displayed within a Frame have their own origin and scalefactor. Manipulations applied to the Frame are passed on to the sub-actuators. For example if the Frame is set as invisible or non-selectable, all of the sub-actuators become invisible or non-selectable.

List_act

List_Act is a sub-actuator which is used in two standard Actuators: Dirview and Listview. The List_Act List pointer is a linked list of user-defined items that is constructed with the ToolBox functions create_list() and create_node().

Listview

Listview is a compound actuator that provides a means to view a list of user-defined items. The default dimensions of the Listview are 10 lines (items) and 25 columns. The currently selected item is displayed at the top of the Listview and is stored in the disp_name field. The add_item typein is used to add a new item to the list. The accept button is copies the currently selected item to the user-defined target.

Menu

Menu is a basic actuator that provides a means to make a selection. The default layout of a menu is one row and six columns, and this can be modified as necessary. The cell size as well as the font factor can be modified.

Meter

Meter is a compound actuator that provides a means to display output. Types include an arc meter, a dial meter, a horizontal bar meter and a vertical bar meter. The face of the meter can be customized in terms of the number of tics and their size, and the size of the mark. The limits of the meter can be displayed, and their format can be specified. The value for each type can be displayed as a filled meter or as a standard mark. A damping factor can be specified for all meter types.

Scroll

Scroll is a compound Actuator that groups a set of sub-actuators and provides the means to scroll a relatively small display area across a larger Frame containing the sub-actuators. Internally, the Scroll behaves the same as a Frame with the addition of controlling sliders.

Slider

Slider is a basic actuator that provides continuous control of a value between two user-specified limits. The current value is indicated by the position of the slider bar within the rectangular body. The two types sliders provided are bar and strip. Orientation of the body and control axis can be vertical or horizontal.

Slideroid

Slideroid is basic Actuator which provides continuous control of a single numeric value. It allows differential control and absolute control of the value based on the region that is selected with the left-mouse. The value display includes five significant figures in the mantissa and a two figure exponent. A reset region resets the value to the User specified initval. A set target region copies the current value to a specified target if defined.

Stripchart

Stripchart is a basic actuator that provides a means to display a running history of values. Up to two pens can run simultaneously on each chart.

Title

Title is a basic actuator that provides a means to display a string of characters. The font size scales proportionately to the height of the title. The width is adjusted automatically to ensure uniformity. Foreground and background colors can be modified.

Typein

Typein is a basic actuator that provides a means to accept input from the keyboard. Multiple typeins are allowed, with each typein operating independently.

Typeout

Typeout is a compound actuator that provides the user a means to view any text-based output. The default dimensions of the text buffer are 5 lines and 80 columns, with lines exceeding the width of the text buffer automatically wrapped. A user specified size limits the buffer size of the displayed text (see Buffer_Act).

Specific Details of Each Actuator

The following sub-section describes each actuator including the detail structure, method of creation, appearance, function and use. The descriptions summarize any special functions provided to access or control an actuator.

Box

```
typedef struct box_type {              /* BOX actuator detail              */
    long      line_width;              /* Box outline line width           */
    long      frgnd_color;             /* Box outline color                */
    long      bkgnd_color;             /* Box background color             */
} Box;
```

Creation:

```
create_actuator(box);
```

Description:

Box provides a rectangular box with a user defined line width, foreground color and background color. It may be used to visually group related Actuators, bound display areas, divide Panels, etc.

Appearance:

A rectangular polygon drawn with the line width specified in pixels, line color as specified by the frgnd_color index, and background fill color as specified by the bkgnd_color index. Default line_width is 2. Default frgnd_color is ACT_BORDER (black). Default bkgnd_color is CLEAR (no fill is drawn).

Function: not selectable.

Value: none.

Special Functions: none.

Notes: none.

see also: Frame.

Buffer_Act

```
typedef struct buffer_act_type {        /* BUFFER_ACT actuator detail          */
    long        mode;                   /* Buffer_Act mode of operation        */
    char        *buf;                   /* text to be displayed                */
    char        *delimstr;              /* auto 'word' selection delimiters    */
    long        start;                  /* first char to display (in upper-left) */
    long        dot;                    /* insertion point                     */
    long        mark;                   /* other end of selection region       */
    long        col, lin;               /* columns & lines in char positions   */
    long        len;                    /* number of chars in buffer           */
    long        size;                   /* buffer size in bytes                */
    Coord       ch;                     /* character height in pixels          */
    Coord       cw;                     /* character width                     */
    Coord       cd;                     /* character descender                 */
    float       newval;                 /* most recently set value for buffer_act */
} Buffer_Act;
```

Creation:

```
create_actuator(buffer_act);
```

Description:

Buffer_Act is a sub-actuator which is used in two standard Actuators: Fileview and Typeout. The Buffer_Act takes a character string, or buffer, and displays it using a fixed font size of 12 point and non-proportional spacing. Carriage returns and line feeds are treated as new lines, and the default width is 80 columns. The Buffer_Act provides six modes of control: BUF_ACT_FIXED constrains the buffer to a maximum size, BUF_ACT_FREE allows an unlimited buffer size, BUF_ACT_NORMAL provides a cursor and allows text in the buffer to be high-lighted, BUF_ACT_NOCURSOR allows text to be high-lighted but does not provide a cursor, BUF_ACT_NOREGION provides a cursor but does not allow text to be high-lighted, and BUF_ACT_NOCONTROL provides only the buffer with no cursor and no high-lighting of text. The variable delimstr can be used to specify delimiting characters so that the cursor moves automatically from string to string.

Buffer_Act functional modes specified may be combined using 'or':

BUF_ACT_FIXED	0x01
BUF_ACT_FREE	0x02
BUF_ACT_NORMAL	0x04
BUF_ACT_NOCURSOR	0x08
BUF_ACT_NOREGION	0x10
BUF_ACT_NOCONTROL	
(BUF_ACT_NOCURSOR | BUF_ACT_NOREGION)	

Appearance:

A Buffer_Act is rendered as a rectangle with a negative bevel. Text is black against an off-white background. The cursor is dark-blue, with underlying text intense white. High-lighted text is intense white on a light-blue background.

Function:

The left-mouse is used to place the cursor in the text region. Holding down the left-mouse and moving it high-lights text in the region. If the left-mouse is held down and moved out of the text region, either above or below, the text will scroll in the corresponding direction.

Value: none.

Special Functions:

add_to_buffer(Actuator*, char*) - appends text referenced by second argument to end of buffer.

buffer_window_down(Actuator*) - move the Buffer_Act viewing window down one line of text at a time causing text to appear to scroll up.

buffer_window_up(Actuator*) - move the Buffer_Act viewing window up one line of text at a time causing text to appear to scroll down.

copy_buffer_all(Actuator*, char*) - copy the entire contents of the Buffer_Act buffer to the destination referenced by the second argument User must ensure that destination memory allocation is large enough.

copy_buffer_block(Actuator*, char*) - copy the contents of the high-lighted block of text from the Buffer_Act buffer to the destination referenced by the second argument User must ensure that destination memory allocation is large enough.

load_buffer(Actuator*, char*) - load Buffer_Act buffer with text referenced by second argument replacing any previously existing text in the buffer.

Notes: none.

see also: Fileview, Typeout.

Button

```
typedef struct button_type {          /* BUTTON actuator detail          */
    long      btype;                  /* Button sub-type                 */
    long      shape;                  /* Button shape                    */
    long      symbol;                 /* Symbol if defined               */
    float     orientation;            /* Button symbol orientation       */
} Button;
```

Creation:

```
create_actuator(button);
create_actuator(simple_button);
create_actuator(toggle_button);
create_actuator(radio_button);
create_actuator(arrow_button);
create_actuator(double_arrow_button);
create_actuator(label_button);
```

Description:

Button is a basic Actuator which has two states: 'ON' or 'OFF'. The ToolBox provides three functional sub-types of button: simple which is 'ON' only when selected with the left-mouse, toggle which swaps state with each left-mouse selection, and radio which forms into a group with only the most recently selected button 'ON' and the others 'OFF'.

Button functional sub-types (btype) are specified:

BUTTON_SIMPLE	21
BUTTON_TOGGLE	22
BUTTON_RADIO	23

Appearance:

A Button may be rendered as a circle or a rectangle as specified in shape:

CIRCLE	0
RECTANGLE	1

A Button may display one of several symbols on its face as specified by symbol:

NO_SYMBOL	0
TOGGLE	1
ARROW	2
SINGLE_ARROW	2
DOUBLE_ARROW	3
LABEL	4

TOGGLE presents a highlighted X for rectangular Buttons and 'spot' for circular Buttons which appears only when the Button is 'ON'. SINGLE_ARROW and DOUBLE_ARROW present a single or double triangle as a directional symbol which appears in an inverse color scheme when the Button is 'ON'. LABEL presents the text of the Actuator label centered on the face and highlighted when the Button is 'ON'.

Orientation applies only to a Button with SINGLE_ARROW or DOUBLE_ARROW specified for its symbol.

ARROW_UP	0.0
ARROW_RIGHT	90.0
ARROW_DOWN	180.0
ARROW_LEFT	270.0

Function:

A simple Button functions as a momentary contact switch which is 'ON' while selected with the left-mouse and 'OFF' at all other times. A toggle Button retains its state and swaps it ('OFF' to 'ON' or vice versa) once per selection.

A radio Button must be combined with other radio Buttons all having identical group_ids to form a group. After creating one or more sets of radio Buttons having common group_ids and adding them to a Panel, call reset_groups() to properly link the groups together. An additional radio Button may be added to an existing group by setting its group_id to that of the group and calling add_to_group(). A radio Button can be removed from its group by calling remove_from_group().

Value:

Button val is maxval when 'ON' and minval when 'OFF'.

Special Functions:

is_button_on(Actuator*) - returns state of the specified Button, TRUE if 'ON' and FALSE if 'OFF'.

Notes: none.

see also: group_id, reset_groups(), add_to_group(), remove_from_group().

Cycle

```
typedef struct cycle_type {          /* CYCLE actuator detail                */
    Actuator  *frame,                /* Cycle surrounding frame              */
              *member_list,          /* Cycle member list                    */
              *prior,                /* reference to prior shift button      */
              *next;                 /* reference to next shift button       */
} Cycle;
```

Creation:

```
create_actuator(cycle);
```

Description:

Cycle is a compound Actuator which displays one of a set of sub-actuators that have been added to it. The User may advance through the list of sub-actuators by selecting the cycle body outside of the displayed sub-actuator.

Appearance:

The Cycle is rendered as a rectangular enclosure large enough to surround the current sub-actuator. Only the current sub-actuator is displayed within the Cycle frame. The Cycle adjusts to accommodate changes in current sub-actuator size as it is advanced.

Function:

Sub-actuators are selected in the normal fashion. The current sub-actuator is advanced when the Cycle is selected with the left-mouse button while the mouse-cursor is outside of the current sub-actuator.

Value: assigned the value of the most recently actuated sub-actuator.

Special Functions:

add_member_to_cycle(Actuator* sa, Actuator* parent) - Add the sub-actuator specified in the first argument to the parent Cycle specified in the second argument. Both the Cycle and sub-actuator are fixed after the addition.

Notes: none.

see also: Frame, Scroll.

Dial

```
typedef struct dial_type {          /* DIAL actuator detail                    */
    long    shape;                  /* Shape {CIRCLE or RECTANGLE}             */
    Coord   r;                      /* Dial Radius                             */
    long    major_tics;             /* Number of major tics                    */
    long    minor_tics;             /* No. of minor tics between each major    */
    float   tl, tw;                 /* Tic mark length and width               */
    float   ml, mw;                 /* Indicator mark length and width         */
    float   theta, thetaset;        /* Theta current value and reset value     */
    float   mintheta, maxtheta;     /* Theta min and max for dial              */
    long    mode;                   /* Mode: wrap values or block ends         */
    float   winds;                  /* Number of revolutions min to max        */
    float   reference;              /* Reference position for fine control     */
    float   finefactor;             /* Fine control factor                     */
} Dial;
```

Creation:

```
create_actuator(dial);
```

Description:

Dial is a basic Actuator which has two modes as specified in mode:

DIAL_WRAP	0x01
DIAL_BLOCK	0x02

Wrap mode allows the value mark to be moved freely around the dial and cross through the minval and maxval end points. Block mode causes the value mark to be restricted in movement to the endpoints. Winds specifies the number of revolutions corresponding to the value range from minval to maxval. Default winds = 1. Finefactor specifies the reduced sensitivity factor to be applies when FineAdjust is selected. Default finefactor = 0.1.

Appearance:

A Dial may be rendered as a circle or a rectangle as specified in shape:

CIRCLE	0
RECTANGLE	1

The number and size of tics are specified by the user. The size of the mark is also specified by the user. The standard diameter of the face is 0.8 times the minimum of the width and height of the dial.

Function:

The dial is activated with the left-mouse. Fine control is achieved either by using the left-mouse and middle-mouse buttons together, or by using the left-mouse with a control key.

Value:

The value of a dial is determined by the mark position between the specified endpoints.

Special Functions: none.

Notes: none.

Dirview

```
typedef struct dirview_type {          /* LISTVIEW actuator detail          */
    Actuator  *dir_list;               /* Directory List_Act pointer        */
    Actuator  *accept;                 /* accept string actuator pointer    */
    Actuator  *reset;                  /* delete string actuator pointer    */
    Actuator  *scroll_bar;             /* Scroll bar actuator pointer       */
    Actuator  *scroll_up;              /* Scroll up arrow actuator pointer  */
    Actuator  *scroll_down;            /* Scroll down arrow actuator pointer*/
    char      file_name[];             /* File name currently chosen        */
    char      disp_file_name[];        /* Displayed filename                */
    char      dir_path[];              /* Full directory path currently chosen */
    char      disp_dir_path[];         /* Displayed (truncated) directory path */
    char      user_filename[];         /* Complete filename for user        */
    char      *target;                 /* Target for entry accept transfer  */
} Dirview;
```

Creation:

```
create_actuator(dirview);
```

Description:

Dirview is a compound actuator that provides a means to view the contents of a directory and select a file. The Dirview actuator controls three types of basic actuators: a List_Act (dir_list), a scroll_bar (scroll_bar), and buttons (accept, reset, scroll_up and scroll_down). The default dimensions of the Dirview are 10 lines (entries) and 25 columns. The width and height of the Dirview actuator can be re-sized. The accept button copies the currently selected item to the user-defined target if the item is a file name, or changes directories if it is a directory name. The reset button returns the Dirview to the current working directory of the user. Item names that exceed the width of the display (e.g. filenames complete with path specification) are truncated for display purposes only, the complete name is stored in the data structure.

Appearance:

A Dirview is rendered as a rectangle with the scroll buttons and scroll bar on the left, the List_Act region in the middle, and the accept and reset buttons along the bottom. The current directory is displayed in a high-lighted box at the top of the Dirview above the List_Act. In the List_Act region, directory entries are black against an off-white background, and the currently selected entry is intense white on a light-blue background.

Function:

The left-mouse is used to place the cursor in the text region, scroll through the items using the scroll buttons and the scroll bar, send the selected item to a target using the accept button if the item is a file name, or else change directories if it is a directory name, and reset the Dirview to the current working directory using the reset button. Holding down the left-mouse and moving it in the text region scrolls through the list.

If the left-mouse is held down and moved out of the text region, either above or below, the text will scroll in the corresponding direction.

Value: The current entry selection is maintained in the user_filename field.

Special Functions:

copy_dirview_entry(Actuator*, char*) - copy the current entry selection from user_filename to the destination specified by the second argument. User must ensure that destination memory allocation is large enough.

Accept Button - copies the current entry selection from user_filename to the destination referenced by target field if entry is a file name, and changes directory if entry is a directory.

Reset Button - resets the Dirview to the current working directory of the user.

Notes: none.

see also: List_Act, Listview

Fileview

```
typedef struct fileview_type {          /* FILEVIEW actuator detail              */
    float       newval;                 /* most recently set value for fileview  */
    Actuator    *f_buffer;              /* buffer for file listing               */
    Actuator    *filename;              /* filename typein actuator pointer      */
    Actuator    *accept;                /* accept string actuator pointer        */
    Actuator    *scroll_bar;            /* Scroll bar actuator pointer           */
    Actuator    *scroll_up;             /* Scroll up arrow actuator pointer      */
    Actuator    *scroll_down;           /* Scroll down arrow actuator pointer    */
    char        *target;                /* Target for entry accept transfer      */
} Fileview;
```

Creation:

```
create_actuator(fileview);
```

Description:

Fileview is a compound actuator that provides a means to select and view a file. The Fileview actuator combines four types of basic actuators: a buffer_act (f_buffer), a typein (filename), a scroll_bar (scroll_bar), and buttons (accept, scroll_up and scroll_down). The default dimensions of the text buffer are 15 lines and 80 columns, with lines exceeding the width of the text buffer automatically wrapped. The width and height of the Fileview actuator can be re-sized. A user specified size limits the buffer size of the displayed file (see Buffer_Act). The accept button is used to copy high-lighted text to the user-defined target.

Appearance:

A Fileview is rendered as a rectangle with the scroll buttons and scroll bar on the left, the buffer_act text region in the middle, and the typein and accept buttons along the bottom. In the text region, text is black against an off-white background, the cursor is dark-blue with underlying text intense white, and high-lighted text is intense white on a light-blue background.

Function:

The left-mouse is used to place the cursor in the text region, scroll through the text using the scroll buttons and the scroll bar, send selected text to a target using the accept button, and load a file using the typein. Holding down the left-mouse and moving it high-lights text in the region. If the left-mouse is held down and moved out of the text region, either above or below, the text will scroll in the corresponding direction.

Value: none.

Special Functions:

copy_fileview_block(Actuator*, char*) - copy the contents of the high-lighted block of text from the buffer to the destination referenced by the second argument User must ensure that destination memory allocation is large enough.

load_fileview(Actuator*, char*) - load the Fileview buffer with text from the file specified by the second argument.

Notes: none.

see also: Buffer_Act, Typeout.

Frame

```
typedef struct frame_type {          /* FRAME actuator detail                */
    long        mode;                /* Mode of Frame operation              */
    Coord       offx, offy;          /* Origin offset of Frame display       */
    Coord       minx, maxx,          /* Bounding box for all sub-actuators   */
                miny, maxy;          /* within Frame.                        */
    Coord       margin;              /* margin for Frame bounding box        */
} Frame;
```

Creation:

```
create_actuator(frame);
```

Description:

Frame is a compound Actuator used to group and display a set of sub-actuators. Actuators added to and displayed within a Frame have their own origin and scalefactor. Manipulations applied to the Frame are passed on to the sub-actuators. For example if the Frame is set as invisible or non-selectable, all of the sub-actuators become invisible or non-selectable.

Appearance:

The Frame is rendered as a rectangular enclosure with a negative bevel. Any actuators visible within the limits of the Frame are drawn.

Function:

Sub-actuators are selected and controlled in a normal fashion. The Frame itself may be assigned action functions which will be processed after those of the selected sub-actuator.

Value: assigned the value of the most recently actuated sub-actuator.

Special Functions:

add_member_to_frame(Actuator* sa, Actuator* parent) - Add the sub-actuator specified in the first argument to the parent Frame specified in the second argument. Both the Frame and sub-actuator are fixed after the addition.

Notes: none.

see also: Cycle, Scroll.

List_Act

```
typedef struct list_act_type {          /* LIST_ACT actuator detail              */
    Link_list  *List;                   /* Specified List                        */
    List_node *selected_node;           /* Pointer to selected entry node        */
    char       selected_name[];         /* Name of selected item                 */
    long       selected_item;           /* Sequential number of selected item    */
    long       total_items;             /* Total number of items in list         */
    long       display_lines;           /* Number of entries displayed in list   */
    float      newval;                  /* Most recently set value for list      */
    float      font_factor;             /* Font factor for item display          */
} List_Act;
```

Creation:

create_actuator(list_act);

Description:

List_Act is a sub-actuator which is used in two standard Actuators: Dirview and Listview. The List_Act List pointer is a linked list of user-defined items that is constructed with the ToolBox functions create_list() and create_node().

Appearance:

A List_Act is rendered as a rectangle with a negative bevel. The font size of the displayed items can be defined by the user (font_factor). The text is black against an off-white background. The currently selected item is high-lighted in dark-blue, with underlying text intense white.

Function:

The left-mouse is used to select an item in the text region. If the left-mouse is held down and moved out of the text region, either above or below, the text will scroll in the corresponding direction.

Value: none.

Special Functions:

add_to_list_act(Actuator*, long order, char *item) - insert the item specified by the third argument into the List_Act list in the order specified by the second argument. Order: HEAD 1, TAIL 2, ASCENDING 3, and DESCENDING 4.

copy_list_act_entry(Actuator*, char*) - copy the currently selected item from selected_name to the destination specified by the second argument. User must ensure that destination memory allocation is large enough.

initialize_list_act(Actuator*, Link_list*) - initialize the List_Act with a user constructed linked list of items specified by the second argument.

remove_selected_entry(Actuator*) - remove the currently selected item from the List_Act linked list.

Notes: none.

see also: Dirview, Listview, create_node(), create_list().

Listview

```
typedef struct listview_type {          /* LISTVIEW actuator detail          */
    Actuator  *lv_list;                 /* Listview list pointer             */
    Actuator  *add_item;                /* Add Item Typein actuator pointer  */
    Actuator  *accept;                  /* accept string actuator pointer    */
    Actuator  *delete;                  /* delete string actuator pointer    */
    Actuator  *scroll_bar;              /* Scroll bar actuator pointer       */
    Actuator  *scroll_up;               /* Scroll up arrow actuator pointer  */
    Actuator  *scroll_down;             /* Scroll down arrow actuator pointer*/
    char disp_name[];                   /* Item name currently chosen        */
    char      *target;                  /* Target for entry accept transfer  */
} Listview;
```

Creation:

```
create_actuator(listview);
```

Description:

Listview is a compound actuator that provides a means to view a list of user-defined items. The Listview actuator combines four types of basic actuators: a List_Act (lv_list), a typein (add_item), a scroll_bar (scroll_bar), and buttons (accept, delete, scroll_up and scroll_down). The default dimensions of the Listview are 10 lines (items) and 25 columns. The width and height of the Listview actuator can be re-sized. The currently selected item is displayed at the top of the Listview and is stored in the disp_name field. The add_item typein is used to add a new item to the list. The accept button is copies the currently selected item to the user-defined target. Item names that exceed the width of the display (e.g. filenames complete with path specification) are truncated for display purposes only, with the complete name stored in the data structure.

Appearance:

A Listview is rendered as a rectangle with the scroll buttons and scroll bar on the left, the List_Act region in the middle, the typein above the List_Act, and the accept and delete buttons along the bottom. The current item is displayed in a high-lighted box above the List_Act. In the List_Act region, the listed items are black against an off-white background, and the currently selected item is intense white on a light-blue background.

Function:

The left-mouse is used to place the cursor in the text region, scroll through the items using the scroll buttons and the scroll bar, send the selected item to a target using the accept button, delete the selected item from the list, and add an item using the typein. Holding down the left-mouse and moving it in the text region scrolls through the list. If the left-mouse is held down and moved out of the text region, either above or below, the text will scroll in the corresponding direction.

Value: The currently selected item is also maintained in the disp_name field.

Special Functions:

add_to_listview(Actuator*, long order, char *item) - insert the item specified by the third argument into the Listview linked list in the order specified by the second argument. Order: HEAD 1, TAIL 2, ASCENDING 3, and DESCENDING 4.

copy_listview_entry(Actuator*, char*) - copy the currently selected item from disp_name to the destination specified by the second argument. User must ensure that destination memory allocation is large enough.

load_listview(Actuator*, Link_list*) - load the Listview with the user constructed linked list specified by the second argument.

Notes: none.

see also: List_Act, Dirview

Menu

```
typedef struct menu_type {              /* MENU actuator detail                 */
    long    cols, rows;                 /* Number of rows and columns in menu */
    float   cell_width, cell_height;    /* Width and height of menu cells       */
    long    menu_choice;                /* Most recently selected menu choice   */
    long    on_color, off_color;        /* Label colors when on and off         */
    float   font_factor;                /* Font factor for labels.              */
                                        /* Array of menu labels                 */
    char    labels[MAX_MENU][MAX_STRING_LEN+1];
} Menu;
```

Creation:

```
create_actuator(menu);
```

Description:

Menu is a rectangular display of user-defined selections. The number of rows and columns can be customized. The default dimensions are six rows and one column. The size of the cells, as well as the font factor for the labels, can be customized.

Appearance:

If the left mouse is held down and moved over the cells, the cell beneath the cursor is high-lighted, as specified by the on_color and off_color fields.

Function:

The left mouse is used to select a menu item. The cursor is placed over the desired selection and the left mouse is pressed and released. The menu selection is returned on the up transition.

Value:

The menu_choice field holds the index of the most recently selected menu item.

Special Functions: none.

Notes: none.

see also:

Meter

```
typedef struct meter_type {              /* METER actuator detail                  */
    long        mtype;                   /* Meter type                             */
    Coord       r;                       /* Dial and arc meter Radius              */
    long        major_tics;              /* Number of major tics                   */
    long        minor_tics;              /* No. of minor tics between major tics   */
    float       tl, tw;                  /* Tic mark length and width              */
    float       ml, mw;                  /* Indicator mark length and width        */
    long        mcolor;                  /* Indicator mark color                   */
    float       thetaset;                /* Relative zero position of meter        */
    Actuator    *high_limit;             /* High limit title actuator              */
    Actuator    *low_limit:              /* Low limit title actuator               */
    Boolean     display_limits;          /* Display minval and maxval limits       */
    char        limits_fmt[MAX_FMT_LEN+1]; /* limits display format str            */
    long        damping factor;          /* Number of past values to average       */
    long        history_ndx;             /* Current beginning of history           */
    float       *history;                /* Series of meter values for damping     */
} Meter;
```

Creation:

```
create_actuator(meter);
create_actuator(arc_meter);
create_actuator(filled_arc_meter);
create_actuator(dial_meter);
create_actuator(filled_dial_meter);
create_actuator(vbar_meter);
create_actuator(vstrip_meter);
create_actuator(hbar_meter);
create_actuator(hstrip_meter);
```

Description:

Meters are output devices designed to graphical display numerical data.

Appearance:

Standard arc meters are rendered as rectangles with a half-circle shaped inverted dial. Arc dial meters are rendered as rectangles with a complete circle enclosed. Bar meters are rendered as rectangles with the value mark moving along the length of the actuator. Standard meters have a single mark to indicate the current value. Strip and filled meters fill the meter from the initial value to the current value with a distinct color. The type of meter is specified in the mtype field:

METER_ARC	111
METER_ARC_FILLED	112
METER_DIAL	113
METER_DIAL_FILLED	114
METER_VBAR	115

METER_VSTRIP	116
METER_HBAR	117
METER_HSTRIP	118

Limits can be displayed in a specified format, and meter faces can be customized in terms of the number and size of tic marks, the size and color of the mark, and the radius of the dial for arc and dial meters.

Function:

The meter is driven by the application. Damping can be induced using the damping_factor field.

Value: none (the meter is an output device).

Special Functions:

set_meter_value(Actuator*, float) - Record the value specified in the second argument into the meter's history array and val attribute. Set the need for redraw.

Notes: none.

see also:

Scroll

```
typedef struct scroll_type {          /* SCROLL actuator detail              */
    Actuator  *v_scroll,              /* Vertical scroll bar controller      */
              *h_scroll,              /* Horizontal scroll bar controller    */
              *cabinet,               /* Surrounding cabinet frame           */
              *display;               /* Display frame                       */
} Scroll;
```

Creation:

```
create_actuator(scroll);
```

Description:

Scroll is a compound Actuator that groups a set of sub-actuators and provides the means to scroll a relatively small display area across a larger Frame containing the sub-actuators. Internally, the Scroll behaves the same as a Frame with the addition of controlling sliders.

Appearance:

The Scroll is rendered as a cabinet Frame having a negative bevel with a display window Frame inside also having a negative bevel. To the left and below the display is a vertical and a horizontal slider respectively. Sub-actuators are rendered within the display Frame.

Function:

Only those sub-actuators visible through the display Frame are selectable and operational. The control Sliders may be used to adjust the location of the display window relative to the larger Scroll area.

Value: Assigned the value of the most recently actuated sub-actuator.

Special Functions:

add_member_to_fscroll(Actuator* sa, Actuator* parent) - Add the sub-actuator specified in the first argument to the parent Scroll specified in the second argument. Both the Scroll and sub-actuator are fixed after the addition.

Notes: none.

see also: Cycle, Frame.

Slider

```
typedef struct slider_type {          /* SLIDER actuator detail              */
    long    stype;                    /* Slider type                         */
    long    mode;                     /* Mode of operation                   */
    float   mw, mh;                   /* Indicator mark length and width     */
    float   mbw;                      /* Indicator mark bevel width          */
    long    mcolor;                   /* Indicator mark color                */
    float   reference;                /* Reference position for fine control */
    float   finefactor;               /* Fine control factor                 */
} Slider;
```

Creation:

```
create_actuator(slider);
create_actuator(vbar_slider);
create_actuator(vstrip_slider);
create_actuator(hbar_slider);
create_actuator(hstrip_slider);
```

Description:

Slider is a basic actuator that provides continuous control of a value between two user-specified limits. The current value is indicated by the position of the slider bar within the rectangular body. The two types sliders provided are bar and strip. Orientation of the body and control axis can be vertical or horizontal.

Appearance:

Sliders are long rectangles with a bar running perpendicular to the longer dimension. The bar slides up and down the length of the slider. No highlighting is made when the bar is selected. Orientation and type of mark is specified in the stype field:

SLIDER_VSTRIP	131
SLIDER_VBAR	132
SLIDER_HSTRIP	133
SLIDER_HBAR	134

The mode of the slider is specified in the mode field:

SLIDER_BLOCK	0x01

Function:

The slider is controlled using the left mouse. Fine control is achieved using the left and middle mouse together, or by pressing the control key and left mouse key simultaneously.

Value:

The value of the slider is the value of the endpoints, set by the user, linearly interpolated by the position of the slider bar within the bounding rectangle of the slider.

Special Functions: none.

Notes: none.

see also:

Slideroid

```
typedef struct slideroid_type {          /* SLIDEROID actuator detail          */
    long        mode;                    /* Mode of operation                  */
    Boolean     reset,                   /* Flag indicating to reset val to initval */
                set_target;              /* Flag indicating to set target variable */
    float       *target;                 /* Address of target variable         */
    float       reference;               /* Reference position for fine control */
    float       finefactor;              /* Fine control factor                */
    float       cw, ch, vir, we;         /* Char width & height, val w & expel w *
} Slideroid;
```

Creation:

```
create_actuator(slideroid);
```

Description:

Slideroid is basic Actuator which provides continuous control of a single numeric value. It allows differential control and absolute control of the value based on the region that is selected with the left-mouse. The value display includes five significant figures in the mantissa and a two figure exponent. A reset region resets the value to the User specified initval. A set target region copies the current value to a specified target if defined.

Appearance:

The Slideroid is a rectangular body displaying a floating point number in exponential format with 5 significant figures. Two controlling regions at the top of the slideroid are indicated by diamond icons, the left one open and the right one filled. The open diamond marks differential control and the filled diamond marks absolute control. Two small regions below the value and labeled with an 'S' and 'R' indicate the set target and reset controls. Selecting and operating any of the four control regions displays that region with high-lighting.

Function:

Selection and actuation of the open icon on the top left of the Slideroid body changes the value at a differential rate proportional to the distance the mouse-cursor is vertically separated from the control region. If the mouse-cursor is above, the value increases and vice versa. Selection and actuation of the filled icon on the top right of the body changes the value by an absolute amount proportional to the distance the mouse-cursor is separated from the control region. Above the region increases the value and below decreases it. Selection of the 'S' control region sends the current value to the target location if one is specified. Selection of the 'R' control region resets the Slideroid to its developer specified initval.

Value:

A floating point value between the minval and maxval.

Special Functions: none.

Notes: none.

see also:

Stripchart

```
typedef struct stripchart_type {          /* STRIPCHART actuator detail          */
    long        stype;                    /* Stripchart type                     */
    long        mode;                     /* Stripchart mode of operation        */
    Actuator    *high_limit;              /* High limit title actuator           */
    Actuator    *low_limit:               /* Low limit title actuator            */
    Boolean     display_limits;           /* Display minval and maxval limits    */
    char        limits_fmt[MAX_FMT_LEN+1]; /* limits display format str          */
    Boolean     Bind_High, Bind_Low;      /* Bind the low and high values?       */
    long        num_pts;                  /* No. of points on stripchart         */
    long        firstpt, lastpt;          /* Index to first and last points      */
    float       *chart_1;                 /* 1st array of stripchart values      */
    float       *chart_2;                 /* 2nd array of stripchart values      */
} Stripchart;
```

Creation:

```
create_actuator(stripchart);
create_actuator(dual_stripchart);
```

Description:

Stripcharts are rectangular plotting regions that are used to display a history of data values. As data values are added to the stripchart, they are plotted along the ordinate. The position along the abscissa reflects the order in which the data values are presented to the stripchart. Stripcharts automatically scroll to display the most recently added data.

Appearance:

Stripcharts are rendered with a light background and a black data pen. A optional second pen is drawn in red. If the limits are displayed, they are placed on the right side of the rectangle on the upper and lower corners. Stripcharts can have one or two pens. The default colors for pens one and two are ALT1 and ALT2, respectively.

Function:

Stripcharts are output devices and thus don't respond to mouse action. If the Bind_High and Low flags are inactive, the stripchart automatically adjusts its high and low limits to maximize vertical resolution.

Value: none (the stripchart is an output device).

Special Functions:

clear_stripchart(Actuator*) - clear and reset the specified stripchart.

set_stripchart_value(Actuator*, float, float) - Add the values specified in the second and third arguments to the chart arrays of the specified Stripchart. If the Stripchart is a single pen Stripchart, the third argument is ignored. Set the need for redraw.

Notes: none.

see also: Meter

Title

```
typedef struct title_type {              /* TITLE actuator detail                */
    long        bkgnd_color;             /* Title background color index         */
    long        frgnd_color;             /* Title foreground color index         */
} Title;
```

Creation:

```
create_actuator(title);
```

Description:

Titles are displays of character strings. They can be static or dynamic, and the color of both the characters and the background can be customized.

Appearance:

Titles are rendered as a character string in a scalable font. The color of the characters is defined as the frgnd_color, and the background is defined as the bkgnd_color.

Function: none

Value: none.

Special Functions: none.

Notes: none.

see also:

Typein

```
typedef struct typein_type {              /* TYPEIN actuator detail                */
    long        mode;                     /* Termination mode selection            */
    long        state;                    /* Typein current activation state       */
    Link_list   *keyboard_buffer;         /* Typein keyboard buffer                */
    char        str[TYPEIN_MAX_LEN+1]; /* Typein string buffer                     */
    char        reset_str[TYPEIN_MAX_LEN+1]; /* Typein reset string buffer         */
    long        max_len;                  /* Maximum buffer length                 */
    float       font_factor;              /* Font for typein. Can be scaled        */
    char        *target;                  /* Target for completed string           */
    void        (*complete_func)();       /* fn called when typein complete        */
} Typein;
```

Creation:

```
create_actuator(typein);
```

Description:

Typeins are used to accept input from the user. When a typein is active, it will accept input if the cursor is over it and its buffer is not full. Multiple typeins can be active simultaneously. When a typein is completed, its contents can be copied to a target destination specified by the user.

Appearance:

Typeins are long rectangles with negative a bevel. When they are active a block cursor is drawn immediately to the right of the last character. Typeins can be resized, and the font size of the characters can be scaled.

Function:

Typeins are activated with the left mouse. They are terminated according to the specified mode:

TYPEIN_MOUSE_ON	1
TYPEIN_MOUSE_OFF	2

TYPEIN_MOUSE_ON mode enables the user to complete a typein either by pressing the return key or left mousing the typein. TYPEIN_MOUSE_OFF mode only allows the user to complete a typein by using the left mouse. When completed, the typein will call the complete_func() once. The user can utilize the target field to specify the location to receive the completed string.

Value: The state field contains the current activation state of the typein:

TYPEIN_CANCEL	-1
TYPEIN_INACTIVE	0
TYPEIN_ACTIVE	1
TYPEIN_COMPLETE	2

Special Functions:

cancel_typein(Actuator*) - Cancel the specified Typein returning the buffer to the contents it had on activation. Typein buffer is not copied to target and completion function is not called.

complete_func(Actuator*) - Optional developer defined function to be called upon completion of Typein entry. This is equivalent to the upfunc of other Actuators and is provided because a Typein may be active for input without being the Selected_Actuator for the panel.

get_typein_string(Actuator*, char*) - Gets the specified Typein's buffer string when the Typein input is complete. Returns the status of the get operation after each call. As long as the Typein is active for input, it returns PENDING. If the Typein is cancelled, it returns CANCEL. If successful, it returns COMPLETE.

load_typein_string(Actuator*, char*) - Loads the specified Typein with the string specified by the second argument.

Notes: none.

see also:

Typeout

```
typedef struct typeout_type {          /* TYPEOUT actuator detail          */
    Actuator  *t_buffer;               /* Typeout buffer                   */
    Actuator  *scroll_bar;             /* Scroll bar actuator pointer      */
    Actuator  *scroll_up;              /* Scroll up arrow actuator pointer */
    Actuator  *scroll_down;            /* Scroll down arrow actuator pointer */
    long       buffer_size;            /* Size in bytes of typeout buffer  */
} Typeout;
```

Creation:

```
create_actuator(typeout);
```

Description:

Typeout is a compound actuator that provides the user a means to view any text-based output. The Typeout actuator combines three types of basic actuators: a buffer_act (t_buffer), a scroll_bar (scroll_bar), and buttons (scroll_up and scroll_down). The default dimensions of the text buffer are 5 lines and 80 columns, with lines exceeding the width of the text buffer automatically wrapped. A user specified size limits the buffer size of the displayed text (see Buffer_Act). The width and height of the Typeout actuator can be re-sized.

Appearance:

A Typeout is rendered as a rectangle with the scrollbuttons and scroll bar on the left and the buffer_act text region in the middle. In the text region, text is black against an off-white background.

Function:

The left-mouse is used to scroll through the text using the scroll buttons and the scroll bar. If the left-mouse is held down and moved out of the text region, either above or below, the text will scroll in the corresponding direction. The default cursor control for the t_buffer is no control. The default buffer size is 1024 bytes.

Value: none.

Special Functions:

add_to_typeout(Actuator*, char*) - appends text referenced by second argument to end of buffer.

load_typeout(Actuator*, char*) - load typeout buffer with text referenced by second argument replacing any previously existing text in the buffer.

Notes: none.

see also: Buffer_Act, Fileview, load_typeout().

ToolBox Function Specifications

The NPS Panel ToolBox provides a complete library of access, processing and control functions related to both panels and actuators. The ToolBox uses no object oriented method of isolating the panel, actuator and supporting data structures, so direct access to all variables is possible. However, we recommend the disciplined use of ToolBox functions rather than direct reference to the data structures themselves. This sections presents the ToolBox functions alphabetized within functional group.

Panel Related Functions

activate(Panel*)
: Activate the specified panel. Set the panel's active state flag to TRUE.

append_panel(Panel*, PanelList*)
: Add the panel specified in the first argument to the tail of the PanelList specified by the second argument. ToolBox calls fix_panel() after the new panel is added.

clear_flag(Panel*, state flag name)
: Clear the specified panel's state flag to FALSE (0).

clear_panel_back(Panel*)
: Clear the backbuffer of the window associated with the specified panel.

clear_panel_both(Panel*)
: Clear both backbuffer and frontbuffer of the window associated with the specified panel.

clear_panel_front(Panel*)
: Clear the frontbuffer of the window associated with the specified panel.

clear_panel_overlay(Panel*)
: Clear the overlay planes of the window associated with the specified panel.

close_panel(Panel*)
: Close the window associated with the specified panel. Close_panel() does not alter the attributes or state values maintained in the panel structure, so if visible is not set to FALSE by the application program, the panel will be re-initialized during the next drawing loop. Set_panel_invisible() is the recommended procedure for temporarily hiding a panel under application program control.

create_panel()
: Create and initialize a new panel setting all default parameters. Customization may be applied to any attributes after creation. Append_panel() or insert_panel() must be used to add the new panel to a PanelList for processing and drawing. If modifications are made after the panel is added to the PanelList, fix_panel() must be called to bind the changes to the panel. Returns a pointer to the new panel.

de_activate(Panel*)
: De-activate the specified panel. Set the panel's active state flag to FALSE.

delete_panel(Panel*)

Remove the specified panel from its associated PanelList, deleting all actuators contained on the panel and freeing all associated memory.

draw_all_panels(PanelList)

Traverse the specified PanelList, drawing each panel which is visible and either has at least one changed actuator or has been set for redraw. Non-visible and unchanged panels are not draw.

draw_panel(Panel*)

If the specified panel is defined, visible and either changed or set for redraw, the ToolBox sets the associated window as active and draws those actuators which have changed or have been set individually for redraw. If the panel is set for redraw, then the background is redrawn first followed by all of the actuators. If the panel is not selectable, a non-selectable cross-hatching is drawn over the panel and its actuators.

draw_panel_background(Panel*)

If the specified panel is defined and visible, the ToolBox sets the associated window as active, clears the panel background to the background color and calls bkgndfunc, the developer defined background function, if specified.

dump_panel(Panel*)

If the panel is defined, the ToolBox dumps all of the attributes of the panel followed by all of the attributes and details of all the actuators on that panel to a ascii text file 'panel.txt'.

fix_panel(Panel*)

Bind changes to the specified panel. If it has been set to not visible, then the ToolBox closes the associated window. If it is defined and visible, but no window has been created, the ToolBox creates and initializes the window. If it is defined and visible, the ToolBox calls the fixfunc referenced in the panel structure to bind any attribute changes to the panel. Finally, the ToolBox sets the panel for redraw.

insert_panel(Panel*, PanelList)

Add the panel specified in the first argument to the head of the PanelList specified by the second argument. ToolBox calls fix_panel() after the new panel is added.

is_active(Panel*)

Returns the state of the specified panel's active attribute, TRUE or FALSE. If TRUE, the panel has been set active for processing the panel action functions.

is_border(Panel*)

Returns the state of the specified panel's border attribute, TRUE or FALSE. If TRUE, the panel's associated window is rendered with a standard IRIS title bar and border.

is_changed(Panel*)

Returns the state of the specified panel's act_redraw attribute, TRUE or FALSE. The act_redraw attribute is set to TRUE whenever any of the actuators on the panel change value or state requiring a redraw.

is_fixed(Panel*)
Returns the state of the specified panel's fixed attribute, TRUE or FALSE. If TRUE, the panel's associated window may be moved but not re-sized.

is_popable(Panel*)
Returns the state of the specified panel's popable attribute, TRUE or FALSE. If TRUE, the panel's associated window may be popped to the top of the displayed set of windows.

is_redraw(Panel*)
Returns the state of the specified panel's redraw_cnt attribute, TRUE or FALSE. If TRUE, the panel has been recorded as in need of complete redraw.

is_screen_relative(Panel*)
Returns the state of the specified panel's screen_relative attribute, TRUE or FALSE. If TRUE, the panel's coordinate system is in screen relative units (pixels). If FALSE, the panel's coordinate system is as specified in the panel structure.

is_selectable(Panel*)
Returns the state of the specified panel's selectable attribute, TRUE or FALSE. If TRUE, the panel and its actuators may be selected using the mouse or key-equivalents.

is_visible(Panel*)
Returns the state of the specified panel's visible attribute, TRUE or FALSE. If TRUE, the panel is visible for selection, processing and drawing. If FALSE, no action is taken with respect to the panel.

is_zbuffer(Panel*)
Returns the state of the specified panel's zbuffer attribute, TRUE or FALSE. If TRUE, the ToolBox initializes and draws the panel in 3 dimensions using the Z-buffer. Advanced interface displays can be designed using this mode.

panelExit()
Graceful exit from ToolBox processing. All panels are properly closed and the overlay planes cleared.

pop_panel(Panel*)
If the specified panel's window is not on top of the displayed set of windows, the ToolBox pops it to the top.

process_panel_functions(Panel*)
Process the specified panel's User defined action functions, if any. If TransitionDown is TRUE, the ToolBox calls downfunc. If the panel is active, the ToolBox calls activefunc. If TransitionUp is TRUE, the ToolBox calls upfunc. The functions are called in the above specified order.

process_panels(PanelList)
Process the selected panel and the selected actuator of that panel, if any. Determines the effect of mouse position, button actuation, and keyboard actuation on the value and state of the selected objects. The actuator newvalfunc is called first followed by the action functions if defined, then the panel action functions, if defined, are called.

push_panel(Panel*)
Push the specified panel's window to the bottom of the displayed set of windows.

setPanel(Panel*)
If the specified panel is defined, an associated window exists and the window is not set, the ToolBox sets the associated window as the current graphics window.

set_flag(Panel*, state flag name)
Set the specified actuator's state flag to TRUE (1).

set_panel(Panel*)
If the specified panel is defined and visible, the ToolBox ensures that the panel is initialized and then calls setPanel() followed by the appropriate viewing matrix initializations.

set_panel_attribute(Panel*, attribute field name, value)
For the specified panel, the ToolBox sets the attribute field specified in the second argument to the value specified in the third argument.

set_panel_invisible(Panel*)
If the specified panel is defined and visible, the ToolBox sets its visible attribute to FALSE and calls fix_actuator(), closing the associated window.

set_panel_location(Panel*, long, long)
Set the x and y coordinates of the specified panel's origin (lower left corner) to the values specified in the second and third arguments respectively.

set_panel_redraw(Panel*)
Record the need for a complete redraw of the specified panel. The panel's redraw_cnt is set to 2 so that it will be redraw once in each display buffer.

set_panel_size(Panel*, long, long)
Set the width and height of the specified panel to the values specified in the second and third arguments respectively.

set_panel_title(Panel*, char*)
Set the specified panel's title to the string specified in the second argument.

set_panel_visible(Panel*)
If the specified panel is defined and not visible, the ToolBox sets its visible attribute to TRUE and calls fix_actuator(), initializing an associated window and setting the panel for redraw.

set_panel_world(Panel*, Coord left, right, bottom, top, near, far)
Set the world coordinate system of the specified panel to the values specified in the second through seventh arguments as indicated in the function prototype.

swapbuffers_panel(Panel*)
If the specified panel is defined and visible, the ToolBox swaps the associated window graphics buffer.

test_flag(Panel*, state flag name)
Returns the specified panel's state flag, TRUE or FALSE.

which_panel(PaneList, short)
Determine which panel is associated with the graphics identifier specified by the second argument.

Actuator Related Functions

ACCESS(Actuator type name, Actuator*, detail field name)
Access the specified detail field of the specified actuator. Actuator type name may be any of the type definitions described in Detailed Actuator Specifications. ACCESS may be used as the left-hand or a right-hand variable of an equation.

activate(Actuator*)
Activate the specified actuator. Set the actuator's active state flag to TRUE.

add_actuator_to_group(Actuator*, Actuator * list)
Add the actuator specified by the first argument to its corresponding group within the actuator list specified by the second argument according to group_id. If the new actuator does not match any of the actuator group_ids of the list, it begins a new group ring by itself.

add_sub_actuator(Actuator* sub-act, Actuator* parent)
Add the specified sub-actuator to the specified parent actuator's sa list. The ToolBox calls fix_actuator() on the parent actuator which binds the changes as well as calling fix_actuator() on the new sub-actuator.

append_actuator(Actuator*, Panel*)
Add the specified actuator to the tail of the specified panel's list of actuators. ToolBox calls actuator() after the new actuator is added.

CENTER_X(Actuator*)
Return the x world coordinate of the center of the specified actuator.

CENTER_Y(Actuator*)
Return the y world coordinate of the center of the specified actuator.

clear_flag(Actuator*, state flag name)
Clear the specified actuator's state flag to FALSE.

create_actuator(specific actuator initialization function)
Create and initialize a new actuator setting all default parameters using the specified initialization function. Customization may be applied to any attributes after creation. Append_actuator() or insert_actuator() may be used to add the new actuator to the desired panel for processing and drawing. Add_sub_actuator() may be used to add the new actuator to a parent actuator instead. If modifications are made after the actuator is added, fix_actuator() must be called to bind the changes to the actuator. Returns a pointer to the new actuator.

create_standard_actuator(long actuator constant)
Create and initialize a new actuator setting all default parameters as specified by the actuator constant. Returns a pointer to the new actuator.

de_activate(Actuator*)
De-activate the specified actuator. Set the actuator's active state flag to FALSE.

de_activate_all(Panel*)
De-activate all actuators linked to the specified panel.

delete_actuator(Actuator*)
Delete the specified actuator from its host panel or parent actuator. The ToolBox ensures that group lists and auto-processing lists remain intact.

delete_all_actuators(Panel*)
Traverse the specified panel's actuator list, deleting each actuator and freeing the associated memory.

draw_actuator(Actuator*)
If defined, visible and in need of a redraw, draw the specified actuator. If it is not selectable, a non-selectable cross-hatch is draw over it.

draw_all_actuators(Panel*)
Traverse the specified panel's actuator list, drawing each actuator that is visible and in need of redraw.

extract_actuator(Actuator*)
Extract the specified actuator from its host panel or parent actuator. The ToolBox ensures that auto-processing lists and group lists remain intact. Returns a pointer to the extracted actuator.

fix_actuator(Actuator*)
Bind changes to the specified actuator. All sub-actuators if any are also fixed. If the actuator has a processfunc defined, then the actuator is inserted into its host panel's automatic processing list. Both label and value display locations are recomputed. Finally, the ToolBox sets the need for a redraw.

get_maxvalue(Actuator*)
Return the specified actuator's maxval.

get_minvalue(Actuator*)
Return the specified actuator's minval.

get_unique_ID()
The ToolBox provides a continuing series of unique object identification numbers beginning with negative one and increasing in the negative direction. Get_unique_ID() returns the next available identification number. This function call is equivalent to the in-line MACRO UniqueID.

get_value(Actuator*)
Return the specified actuator's current value.

insert_actuator(Actuator*, Panel*)
Add the specified actuator to the head of the specified panel's list of actuators. ToolBox calls actuator() after the new actuator is added.

is_active(Actuator*)
Returns the state of the specified actuator's active attribute, TRUE or FALSE. If TRUE, the actuator is active.

is_actuator_on(Actuator*)
Returns whether or not the specified actuator is 'ON' (val != minval) or 'OFF' (val == minval).

is_beveled(Actuator*)
Returns whether or not the specified actuator's has a bevel, TRUE or FALSE.

is_label_on(Actuator*)

Returns whether or not the specified actuator's label is set to be displayed, TRUE or FALSE. Any value other than zero (0) in the l_location attribute field indicates the label will be displayed.

is_mbeveled(Actuator*)

Returns whether or not the specified actuator's has a mark bevel, TRUE or FALSE.

is_redraw(Actuator*)

Returns the state of the specified actuator's redraw_cnt attribute, TRUE or FALSE. If TRUE, the actuator has been recorded as in need of redraw.

is_selectable(Actuator*)

Returns the state of the specified actuator's selectable attribute, TRUE or FALSE. If TRUE, the actuator may be selected using the mouse or key-equivalent.

is_value_on(Actuator*)

Returns whether or not the specified actuator's value is set to be displayed, TRUE or FALSE. Any value other than zero (0) in the v_location attribute field indicates the value will be displayed.

is_visible(Actuator*)

Returns the state of the specified actuator's visible attribute, TRUE or FALSE. If TRUE, the actuator is visible for selection, processing and drawing. If FALSE, no action is taken with respect to the actuator unless it is an automatic actuator which is processed during each ToolBox processing cycle.

PICK(Coord x, y, x1, y1, x2, y2)

Determine if the given world coordinate location (x, y) is within the specified rectangular area.

PICKACT(Actuator*, Coord x, y)

Determine if the pick rectangle of the specified actuator contains the given mouse position (x, y). The pick rectangle includes any bevel defined. Returns a Boolean TRUE or FALSE.

process_actuator_functions(Actuator*)

Process the specified actuator's newvalfunc followed by the User defined action functions, if any. If TransitionDown is TRUE, the ToolBox calls downfunc. If the panel is active, the ToolBox calls activefunc. If TransitionUp is TRUE, the ToolBox calls upfunc. The functions are called in the above specified order.

process_newvalue(Actuator*)

Process the specified actuator after an application program has changed its value. This function is used to ensure that a change in value is reflected by a corresponding change in appearance and function.

RADIUS(Actuator*)

Return the radius of the specified actuator. The radius is computed as one-half the minimum of the width and the height.

remove_actuator_from_group(Actuator*)

Remove the specified actuator from its group ring ensuring the integrity of all group ring links.

reset_groups(Actuator* list)
Traverse the specified linked list and reset all group rings according to matching group_id fields. The ToolBox ensures that the group ring for each actuator in the list is properly set.

set_actuator_label(Actuator*, long location, float label_font_factor, char* label)
Set the specified actuator's label to the string specified in the fourth argument. Set the label location and label_font_factor as specified by the second and third argument.

set_actuator_location(Actuator*, Coord x, y)
Set the x and y world coordinate location of the specified actuator's origin (lower left corner).

set_actuator_size(Actuator*, Coord w, h, bw)
Set the width, height and bevel width of the specified actuator.

set_attribute(Actuator*, attribute field name, value)
For the specified actuator, the ToolBox sets the attribute field specified in the second argument to the value specified in the third argument.

set_detail(Actuator type name, Actuator*, detail field name, value)
Set the specified actuator's detail field to the value specified in the fourth argument. Actuator type name may be any of the type definitions described in Detailed Actuator Specifications.

set_detail_string(Actuator type name, Actuator*, detail string field name, char*)
Set the specified actuator's detail string field to the string specified in the fourth argument. Strcpy is used. Actuator type name may be any of the type definitions described in Detailed Actuator Specifications.

set_flag(Actuator*, state flag name)
Set the specified actuator's state flag to TRUE.

set_label_location(Actuator*, Coord x, y)
Set the x and y location for the specified actuator's label as specified in the second and third arguments respectively. Label location is relative to the actuator's origin.

set_label_size(Actuator*, Coord w, h, bx, by)
Set the width, height, border in x and border in y for the specified actuator's label as specified in the second through fifth arguments respectively. Border values are between the left or bottom of the bounding box and the label string.

set_maxvalue(Actuator*, float)
Set the specified actuator's maxval attribute to the value specified by the second argument.

set_minvalue(Actuator*, float)
Set the specified actuator's minval attribute to the value specified by the second argument.

set_redraw(Actuator*)
Record the need for a redraw of the specified actuator. The actuator's redraw_cnt is set to 2 so that it will be redraw once in each display buffer. The ToolBox sets the host panel's changed_act field to TRUE to ensure drawing of the changed actuator.

set_redraw_all(Panel*)
Set the need for redraw in all actuators of the specified panel.

set_size(Actuator*, Coord w, h, bw)
Set the width, height and bevel width of the specified actuator.

set_target_pointer(Actuator type name, Actuator*, destination pointer)
Set the specified actuator's target pointer to that specified in the third argument. Actuator type name may be any of the type definitions described in Detailed Actuator Specifications. The User must ensure that type of destination pointer and required target correspond.

set_value(Actuator*, float value)
Set the specified actuator's val attribute to the value specified by the second argument limited by the minval and maxval attribute values.

set_value_format(Actuator*, long location, float label_font_factor, char* format)
Set the specified actuator's value display format to the string specified in the fourth argument. Set the value display location and value_font_factor as specified by the second and third argument.

set_value_location(Actuator*, Coord x, y)
Set the x and y location for the specified actuator's value display as specified in the second and third arguments respectively. Value display location is relative to the actuator's origin.

set_value_size(Actuator*, Coord w, h, bx, by)
Set the width, height, border in x and border in y for the specified actuator's value display as specified in the second through fifth arguments respectively. Border values are between the left or bottom of the bounding box and the value string.

test_flag(Actuator*, state flag name)
Returns the specified actuator's state flag, TRUE or FALSE.

UniqueID
The ToolBox provides a continuing series of unique object identification numbers beginning with negative one and increasing in the negative direction. UniqueID returns the next available identification number. This in-line MACRO is equivalent to the function call, get_unique_ID().

which_actuator(Actuator* list, Coord x, y)
Determine which if any actuator in the specified list is 'picked' by the location specified by the second and third arguments. Returns a pointer to the picked actuator and NULL if none is picked.

Color Management Functions

define_color_table(long table, entry_index, float r, g, b, a)
Define the specified entry of the specified color table using the r, g, b and alpha values given by the third through sixth arguments.

set_actuator_color(Actuator*, long index)
Set the current graphics color using the specified actuator's color table and the index specified by the second argument.

set_panel_color(Panel*, long index)
Set the current graphics color using the specified panel's color table and the index specified by the second argument.

Error Handling Functions

FatalError(char*)
Print the specified error message to stderr and exits the program.

Perror(char*)
Print the specified error message to stderr and return to calling function.

Font Control Functions

get_strheight(char* string, fmfonthandle, float font_factor)
Returns the height of the specified string for the specified font and font scale factor.

get_strwidth(char* string, fmfonthandle, float font_factor)
Returns the width of the specified string for the specified font and font scale factor.

initializeFONTS()
Initialize ToolBox fonts.

set_current_font(fmfonthandle)
Set the ToolBox global font to that specified in the argument.

General Functions

ABS(x)
Returns the absolute value of its argument.

draw_string(char* string, Coord x, y, float font_factor, long color_table, color_ndx)
Draw the text string specified in the first argument at the specified location using the specified font factor, color table and color table index.

INTERP(lower, upper, proportion)
Returns the specified proportional interpolation between first and second arguments.

initialize_ToolBox()
Initialize all aspects of the NPS Panel ToolBox. This function must be called before creation, modification, processing or drawing of panels and actuators.

LIMIT(val, lower, upper)
Returns a value limited by the lower and upper bound.

MAX(x, y)
Returns the maximum of two arguments.

MIN(x, y)
Returns the minimum of two arguments.

PROPORTION_OF(val, min, max)
Returns the proportion that the value is of the range specified by the second and third arguments.

List Management Functions

Bottom(Link_list*)
: Returns the bottom data node of the specified linked list stack.

clear_list(Link_list*)
: Remove all list nodes from the specified linked list, re-initializing the header structure and freeing the associated memory.

count_nodes(Link_list*)
: Returns the number of nodes in the specified linked list.

create_list()
: Create the header and control structure for a double linked list.

create_node(Void data)
: Create a linked list node with the specified data structure.

create_str(char*)
: Create a new character string from the argument string.

Current(Link_list*)
: Returns the current node of the specified linked list.

CurrentData(Link_list*)
: Returns a pointer to the data of the current linked list node.

DATA(List_node*)
: Returns a pointer to the data of a specified linked list node.

de_Q(Link_list*)
: Removes the next node from a queue linked list and returns a pointer to its data.

empty_list(Link_list*)
: Returns whether or not the specified linked list is empty. TRUE if empty.

en_Q(Void*)
: Creates an entry in a linked list queue for the specified data item.

free_list(Link_list*)
: Free the memory for all of the structures associated with a specified linked list.

Head(Link_list*)
: Returns a pointer to the first node in a specified linked list.

insert(Link_list* list, long order, Void* data)
: Insert the specified data item into the specified linked list in the specified order.

Next(List_node*)
: Returns a pointer to the node following a specified linked list node.

pop(Link_list*)
: Removes the top node from the specified stack linked list and returns a pointer to its data.

push(Link_list* stack, Void* data)
: Create an entry in the specified linked list stack for the specified data item.

Prior(List_node*)
: Returns a pointer to the node preceding a specified link list node.

Predecessor(List_node*)
Returns a pointer to the node preceding a specified link list node.

search(Link_list* list, char* string)
Search the specified linked list for a data node matching the specified search string. Returns TRUE if found and FALSE otherwise.

Successor(List_node*)
Returns a pointer to the node following a specified link list node.

Tail(Link_list*)
Returns a pointer to the last node in the specified link list.

Top(Link_list*)
Returns a pointer to the first node in the specified link list.

Mouse, Button and Keyboard Functions

activate_keyboard(Link_list)
Activate the keyboard and the specified keyboard buffer for input.

de_activate_keyboard(Link_list)
De-activate the keyboard and the specified keyboard buffer.

FineAdjust {Boolean}
The state of the ToolBox actuator fine-adjustment selection. Fine-adjustment is selected by pressing and holding the middle-mouse button or the Control key while controlling an actuator with the left-mouse button pressed and held.

is_MouseDown {Boolean}
The state of any of the mouse buttons, TRUE if pressed.

Keyboard_Active {Boolean}
The state of the keyboard, TRUE if active and FALSE otherwise.

Process Management Functions

process_actuator_functions(Actuator*)
Process the specified actuator's newvalfunc followed by the User defined action functions, if any. If TransitionDown is TRUE, the ToolBox calls downfunc. If the panel is active, the ToolBox calls activefunc. If TransitionUp is TRUE, the ToolBox calls upfunc. The functions are called in the above specified order.

process_newvalue(Actuator*)
Process the specified actuator after an application program has changed its value. This function is used to ensure that a change in value is reflected by a corresponding change in appearance and function.

process_panel_functions(Panel*)
Process the specified panel's User defined action functions, if any. If TransitionDown is TRUE, the ToolBox calls downfunc. If the panel is active, the ToolBox calls activefunc. If TransitionUp is TRUE, the ToolBox calls upfunc. The functions are called in the above specified order.

process_panels(PanelList)

Process the selected panel and the selected actuator of that panel, if any. Determines the effect of mouse position, button actuation, and keyboard actuation on the value and state of the selected objects. The actuator newvalfunc is called first followed by the action functions if defined, then the panel action functions, if defined, are called.

process_ToolBox_Q(PanelList, short TOKdevice, TOKvalue)

Provides ToolBox processing of each event token. This function must be place inside the event token processing loop and should be placed before the application switch statement that process event tokens. The working PanelList is specified by the first argument. The device id and value are passed to process_ToolBox_Q via the second and third arguments.

reset_ToolBox_Q()

Prepare the ToolBox for each graphics device event queue processing cycle. This function must be placed prior to the event token processing.

Screen Management Functions

clear_screen_overlay()

Clear the overlay planes for the entire screen display area.

System Level Support Functions

tbx_calloc(long size)

Returns a pointer to a memory allocation of the specified size. Sets all memory locations to NULL. If unable to complete the allocation, the ToolBox prints an appropriate error message and exits the program.

tbx_malloc(long size)

Returns a pointer to a memory allocation of the specified size. If unable to complete the allocation, the ToolBox prints an appropriate error message and exits the program.

tbx_realloc(void*, long size)

Returns a pointer to a memory re-allocation of the specified size. If unable to complete the allocation, the ToolBox prints an appropriate error message and exits the program.

ToolBox Constants, Global Variables and Support Structures

The NPS Panel ToolBox provides a complete library of access, processing and control constants and global variables related to both panels and actuators. The ToolBox uses no object oriented method of isolating the panel, actuator and support data structures, so direct access to all constants and global variables is possible. However, we recommend the disciplined use of ToolBox constants and global variables rather than direct reference to the data structures themselves. This sections presents the ToolBox constants and global variables alphabetized within functional group.

Panel Related Constants, Global Variables and Support Structures

Current_Panel {Panel*}
Global pointer to current panel data structure.

mx_Current {Screencoord}
Global reference to x screen coordinate for the current mouse position.

mx_Ref {Screencoord}
Global reference to x screen coordinate for the mouse at most recent mouse button actuation.

my_Current {Screencoord}
Global reference to y screen coordinate for the current mouse position.

my_Ref (Screencoord}
Global reference to y screen coordinate for the mouse at most recent mouse button actuation.

Panel_List {PanelList}
Global double linked list of all panels within a ToolBox supported application.

wx_Current {Screencoord}
Global reference to x coordinate for the current mouse position converted to Panel coordinate system.

wx_Ref {Screencoord}
Global reference to x coordinate for the mouse at most recent mouse button actuation converted to Panel coordinate system.

wy_Current {Screencoord}
Global reference to y coordinate for the current mouse position converted to Panel coordinate system.

wy_Ref (Screencoord}
Global reference to y coordinate for the mouse at most recent mouse button actuation converted to Panel coordinate system.

Actuator Related Constants, Global Variables and Support Structures

x_Ref {Coord}
Global reference to x coordinate (in Panel units) of the current actuator's origin.

y_Ref {Coord}
Global reference to y coordinate (in Panel units) of the current actuator's origin.

Current_Actuator {Actuator*}
Global pointer to current actuator's data structure.

MAX_STR_LEN = 128
Maximum number of characters in ToolBox strings.

MAX_FMT_LEN = 16
Maximum number of characters in value format string.

MAX_BUF_LEN = 256
Maximum number of characters in default buffers.

Selected_Actuator {Actuator*}
Global pointer to currently selected actuator if any.

Color Management Constants, Global Variables and Support Structures

Color_Table[MAX_COLOR_TABLES][MAX_COLORS]
The ToolBox color tables.

Color Table Index Constants:

CLEAR	= -1
PANEL_CLEAR	= CLEAR
ACT_CLEAR	= CLEAR
PANEL_BKGND	= 8
PANEL_LIGHT	= 9
ACT_LIGHT	= 9
PANEL_NORM	= 10
ACT_FACE	= 10
PANEL_ALT	= 11
ACT_BODY	= 11
PANEL_HI	= 12
ACT_DARK	= 12
PANEL_BORDER	= 13
ACT_BORDER	= 13
BEVEL_LIGHT	= 14
PANEL_DARK	= 15
BEVEL_DARK	= 15
MARK_LIGHT	= 16
MARK_DARK	= 17
PANEL_LABEL	= 18
ACT_LABEL	= 18
TYPEIN_BKGND	= 19
PANEL_INPUT	= 19
TYPEIN_CURSOR	= 20

PANEL_CURSOR = 20
ACT_CURSOR = 20
ALT_COLOR_1 = 21
ALT_COLOR_2 = 22
ALT_COLOR_3 = 23

MAX_COLORS = 24
Defines the number of color entries in each color table.

MAX_COLOR_TABLES = 8
Defines the number of color tables.

Font Control Constants, Global Variables and Support Structures

Current_font
ToolBox reference to most recently selected font.

Current_Font_Factor {float}
ToolBox reference to most recently font scale factor. Used to prevent repeat application of the same scale factor thereby increasing ToolBox efficiency.

font_base = Times-Roman 1.0 point
Pointer to ToolBox basic font. ToolBox uses IRIS font manager scaling and rendering functions.

font_base_bold = Times-Bold 1.0 point
Pointer to ToolBox basic bold font. ToolBox uses IRIS font manager scaling and rendering functions.

General Constants, Global Variables and Support Structures

BEL = '\007'
Bell character

Boolean = long
Type definition used for logical operations.

BS = '\010'
Backspace character

Colorndx = long
Type definition used to index color table.

CR = '\015'
Carriage return character

CTRL_C = '\003'
Control-C character

CTRL_U = '\025'
Control-U character

DEL = '\177'
Delete character

EOS = '\000'
End of String character

ESC = '\033'
Escape character

HT = '\011'
Horizontal tab character

LF = '\012'
Line feed character

NUL = '\000'
Null character

Screen = short
Type definition alternate for Screencoord.

SPC = ' '
Space character

TBX_EOF = -1
End of Buffer/File flag

Void = char
Type definition for data structure pointers.

List Management Constants, Global Variables and Support Structures

Link_list
Linked list data structure type definition. Supports double linked lists of List_nodes.

List_node
Data node structure definition. Supports any type of data within a linked list.

Order of Insertion Constants (used by insert() function):

HEAD	1
TAIL	2
ASCENDING	3
DESCENDING	4

Mouse, Button and Keyboard Related Constants and Global Variables

AltKey {Boolean}
Records the state of either Alt key pressed as TRUE.

ControlKey {Boolean}
Records the state of either Control key pressed as TRUE.

Keyboard_Buffer {Link_list*}
Global pointer to ToolBox general keyboard buffer. May be used to accept input from the keyboard if no Panel or Typein keyboard buffers are defined.

Keyboard_State {Boolean}
Records whether or not the keyboard has been activated. Activated = TRUE.

KeyButton {Boolean}
State of any keyboard button, UP or DOWN.

LeftMouse {Boolean}
State of the left-mouse button, UP or DOWN.

MiddleMouse {Boolean}
State of the middle-mouse button, UP or DOWN.

Mouse and Button position constants:
UP = 0
DOWN = 1

MouseButton {Boolean}
State of any mouse button, UP or DOWN.

RightMouse {Boolean}
State of the right-mouse button, UP or DOWN.

ShiftKey {Boolean}
Records the state of either Shift key pressed as TRUE.

TransitionDown {Boolean}
Records any mouse button transition from UP to DOWN as TRUE.

TransitionUp {Boolean}
Records any mouse button transition from DOWN to UP as TRUE.

APPENDIX C

NPS PANEL DESIGNER AND TOOLBOX RESERVED WORDS

Reserved Words for the Intermediate File Parser

The following list of words are reserved for use by the intermediate file parser. They may not be used as the title of a panel. The parser is case-insensitive.

box
buffer_act
button
comment
custom_colors
cycle
dial
dirview
fileview
file_end
frame
listview
menu
meter
panel
panel_end
panel_designer_file
scroll
slider
slideroid
stripchart
title
typein
typeout

APPENDIX D

NPS PANEL DESIGNER AND TOOLBOX SAMPLE GENERATED CODE

The following three files are examples of the code generated by the Code Manager. The filename specified was User_Panel.

```
/*****************************************************************************
*  File:     User_Panel.c                    Prototype panel output code template   *
*  Version:  1.0                                                                    *
*  date:     90/11/17                                                               *
*  Author:   Richard M. Prevatt                                                     *
*            David M. King                                                          *
*  ------------------------------------------------------------------------------- *
*  Notes:                                                                           *
*            90/08/13  Created.                                                     *
*  ------------------------------------------------------------------------------- *
*  This file contains all the functions needed to display and control the           *
*  User defined panel created within PanelDesigner using the Panel Toolbox.        *
*  Appropriate declarations and function calls are included.                        *
*  It is used in conjunction with User_Panel_fn.c                                   *
*                                                                                   *
*  The actual name of this and the related files was derived from the               *
*  name of the current workspace when it was produced by PanelDesigner.            *
*  { Substitute the actual name for 'User_Panel' in these instructions. }           *
*                                                                                   *
*  If a file by that name already existed, the PanelDesigner saved the              *
*  the original version in a backup file as follows:                                *
*   User_Panel.c  -->  User_Panel.c.bak                                             *
*                                                                                   *
*  Compile as follows:                                                              *
*                                                                                   *
*   cc -o user_name User_Panel.c User_Panel_fn.c /nps_path/lib/ npspanel.a           *
*    -I/nps_path/include -O2 -align16 -G 0 -lc_s -lgl_s -lfm  -lm                   *
*                                                                                   *
*            /nps_path must be defined as the proper path to the NPS Panel ToolBox  *
*                                                                                   *
*            /nps_path = /n/gravy1/work/zyda/npspanel in the current release        *
*                                                                                   *
*  The resulting file 'user_name' may be executed.                                  *
*****************************************************************************/

#define     EXTERN                  /* declarations are not external          */
#define     INIT(x)  = (x)          /*   and initialized here                 */

#include    "gl.h"                  /* Graphics Library declarations          */
#include    "device.h"              /* Device declarations                    */

#include    "tbx.h"                 /* Panel Toolbox Declarations             */

/* ---------------------------------------------------------------------------*/
/* User defined and modifiable constants and declarations                     */

#include    "User_Panel.h"

/* ---------------------------------------------------------------------------*/
/* Function Prototypes used within user_panel.c                               */

void        initialize_main();      /* initialize panel environment           */
void        initialize_panels();    /* Initialize the control panels          */
void        initialize_actuators(); /* Initialize the actuators               */
void        initialize_defaults();  /* initialize default settings            */
void        initialize_colors();    /* initialize any user defined colors     */
```

```
void        initialize_queue();          /* Initialize the graphics queue          */
void        initialize_menus();          /* initialize menus here                  */
void        initialize_cursor();         /* Initialize cursors here                */
void        initialize_overlay();        /* Initialize overlay planes here         */
void        control_program              /* Control PanelDesigner operation        */
(
 PanelList  *panel_list                  /*  specified panel list                  */
);
void        process_program_queue();     /* Process graphics event queue           */

void        draw_control_panels          /* Draw user panel and actuators          */
(
 PanelList  *panel_list                  /*  specified panel list                  */
);

/* ------------------------------------------------------------------------------------------*/
        main()
{

 initialize_main();                     /* initialize panel environment           */

 initialize_panels();                   /* Initialize the control panels          */

 initialize_actuators();                /* create the actuators                   */

 /*----- User define initializations are called via user_init_main.               */

 user_init_main();                      /* user define initializations            */

 forever {                              /* Panel main loop                        */

  control_program(Panel_List);          /*  process controls and queue            */

  draw_control_panels(Panel_List);      /*  draw user control panels & acts       */

  /*----- User designed calculations and 2D or 3D drawing functions are          */
  /*-----  accessed via user_display(); User must manage any extra                */
  /*-----  windows required.                                                      */

  user_display();                       /*  handle to call user functions         */
 }
}

/* ------------------------------------------------------------------------------------------*/
void        initialize_main()            /* initialize panel environment           */
{

 initialize_ToolBox();                  /* initialize NPS Panel ToolBox           */

 /*----- initialize all other aspects of main program.                            */

 initialize_defaults();                 /* initialize default settings            */
 initialize_colors();                   /* initialize any user defined colors     */
 initialize_queue();                    /* initialize event graphics queue        */
```

```
 initialize_menus();                    /* initialize PanelDesigner menus          */
 initialize_cursor();                   /* initialize special cursors              */
 initialize_overlay();                  /* initialzie overlay planes & color       */

}

/* ------------------------------------------------------------------------------------*/
void        initialize_panels()         /* Initialize the control panels           */
{
 Panel      *p;                         /* Temporary panel pointer                 */

 /*----- Create each of the user's main control panels.  User may alter           */
 /*-----  any of the parameters of these panels as required, taking               */
 /*-----  care to maintain proper structure and function.                         */

 Control_Panel[0] = p =
    create_panel ();
    set_panel_location(p, 20, 56);
    set_panel_size(p, 720, 534);
    set_attribute(p, visible, TRUE);
    set_attribute(p, popable, FALSE);
    set_attribute(p, fixed, FALSE);
    set_panel_title(p, "User_Panel");
    set_attribute(p, color_table, 1);
    append_panel(p, Panel_List);

}

/* ------------------------------------------------------------------------------------*/
void        initialize_actuators()      /* Initialize the actuators                */
{
 Actuator   *a;                         /* Temporary actuator pointer              */

 /*----- Create each of the actuators required on the control panel.  User        */
 /*-----  may alter the parameters of any actuators as required, taking           */
 /*-----  care to maintain proper structure and function.                         */

 Current_Panel = Control_Panel[0];

 Control[0][0] = a =
    create_actuator(dial);
    set_actuator_location(a, 77.5, 119.5);
    set_actuator_size(a, 75, 75, 2);
    set_attribute(a, group_id, -23);
    set_attribute(a, key, 0);
    set_actuator_label(a, BOTTOM, 10, "Dial");
    set_detail(Dial, a, major_tics, 4);
    set_detail(Dial, a, minor_tics, 0);
    set_detail(Dial, a, winds, 1);
    set_detail(Dial, a, finefactor, 0.1);
    insert_actuator(a, Current_Panel);

 reset_groups(Control_Panel[0]->al_head);

 Current_Actuator(Current_Panel) = NULL;
```

```
 Selected_Actuator = NULL;                /* No actuator selected                */

}

/* ------------------------------------------------------------------------------*/
void          initialize_defaults()      /* Initialize panel defaults           */
{

}

/* ------------------------------------------------------------------------------*/
void          initialize_colors ()       /* Initialize user defined colors      */
{

 /*----- Modify the color tables according to User specifications               */

 define_color_table(0, 8, 0.345, 0.525, 0.835, 0);
 define_color_table(1, 8, 0.345, 0.525, 0.835, 0);
 define_color_table(1, 10, 1, 0, 0, 0);

}

/* ------------------------------------------------------------------------------*/
void          initialize_queue()         /* Initialize the graphics queue       */
{
 user_init_queue();                       /* User defined queue init             */
}

/* ------------------------------------------------------------------------------*/
void          initialize_menus()         /* initialize menus here               */
{
 user_init_menu();                        /* User defined menu init              */
}

/* ------------------------------------------------------------------------------*/
void          initialize_cursor()        /* Initialize extra cursors here       */
{
 user_init_cursor();                      /* User defined cursor init            */
}

/* ------------------------------------------------------------------------------*/
void          initialize_overlay()       /* Initialize overlay planes here      */
{
 user_init_overlay();                     /* User defined overlay init           */
}

/* ------------------------------------------------------------------------------*/
void          control_program            /* Control PanelDesigner operation     */
(
 PanelList    *panel_list                 /*   specified panel list              */
)
{

 process_program_queue();                 /* Process the graphics event queue    */
```

```
 process_panels(panel_list);                    /* Control panels based on user input          */

}


/* ------------------------------------------------------------------------------------------------*/
void        process_program_queue ()           /* Process graphics event queue                */
{
 short      TOKdevice,                         /* Graphics event queue device token           */
            TOKvalue;                          /* Graphics event queue token value            */

 reset_ToolBox_Q();                             /* Prepare ToolBox for input process           */

 while ( qtest() ) {                            /* Process all tokens available                */

  TOKdevice = qread(&TOKvalue);

  /*...................../* Standard ToolBox input processing                                  */

  process_ToolBox_Q(Panel_List, TOKdevice, TOKvalue);

  switch( TOKdevice ) {                         /* User Program specific Q processing          */

   case RIGHTMOUSE:                             /* Right Mouse Controls Menus                  */

    if ( TOKvalue == DOWN )                     /*  on TransitionDown process menu             */
     user_process_menu();                       /* User defined menu processor                 */
    break;

   case ESCKEY:                                 /* Esc key calls for exit                      */

    if ( TOKvalue == UP ) {                     /*  execute when button comes up               */
     user_exit();
    }
    break;

   default:                                     /* Define default processing here              */
    break;

  } /* end switch */

  /*----- User defined queue function receives all TOKENs processed.                            */

  user_process_queue(TOKdevice, TOKvalue);

 } /* end while qtest() */

}
```

```
/* ------------------------------------------------------------------------------------------*/
void            draw_control_panels             /* Draw user panel and actuators          */
(
 PanelList   *panel_list                        /*  specified panel list                  */
)
{
 Panel       *p;

 for ( p = Head(panel_list); p; p = p->next ) {

  draw_panel(p);                                /* Draw all actuators of panel            */

  swapbuffers_panel(p);                         /* swapbuffers in specified panel         */

 }

}


/*****************************************************************************
*  EOF: User_Panel.c          { lines:   310  }                                 *
 *****************************************************************************/
```

```
/*****************************************************************************
*  File:     User_Panel_fn.c          User defined calculations and          *
*  Version:  1.0                      drawing functions                      *
*  date:     90/12/01                                                        *
*  Author:   Richard M. Prevatt                                              *
*            David M. King                                                   *
*  ------------------------------------------------------------------------ *
*  Notes:                                                                    *
*            90/08/13  Created.                                              *
*  ------------------------------------------------------------------------ *
*  This file contains the User modifiable functions needed in support of     *
*  the control panel generated by PanelDesigner.  Changes and additions may  *
*  be added to all files taking care to manage any extra windows.            *
*  It is used in conjunction with User_Panel.c                               *
*                                                                            *
*  The actual name of this and the related files was derived from the        *
*  name of the current workspace when it was produced by PanelDesigner.      *
*  { Substitute the actual name for 'User_Panel' in these instructions. }    *
*                                                                            *
*  If a file by that name already existed, the PanelDesigner saved the       *
*  the original version in a backup file as follows:                         *
*   User_Panel_fn.c  -->  User_Panel_fn.c.bak                                *
*                                                                            *
*  Compile as follows:                                                       *
*                                                                            *
*   cc -o user_name User_Panel.c User_Panel_fn.c /nps_path/lib/ npspanel.a    *
*    -I/nps_path/include -O2 -align16 -G 0 -lc_s -lgl_s -lfm  -lm            *
*                                                                            *
*            /nps_path must be defined as the proper path to the NPS Panel ToolBox *
*                                                                            *
*            /nps_path = /n/gravy1/work/zyda/npspanel in the current release *
*                                                                            *
*  The resulting file 'user_name' may be executed.                           *
*****************************************************************************/

#define     EXTERN   extern          /* declarations are external          */
#define     INIT(x)                  /*   and not initialized here         */

#include    "gl.h"                   /* Graphics Library declarations      */
#include    "device.h"               /* Device declarations                */

#include    "tbx.h"                  /* Panel Toolbox Declarations         */

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

/* User defined and modifiable constants and declarations */

#include    "User_Panel.h"

/* ------------------------------------------------------------------------*/
/* User modifiable function definitions */
```

```
/* ------------------------------------------------------------------------------------------------------------*/
void        user_init_queue()           /* User defined queue init                                          */
{
 /*..... Place user needed event queue device initializations here.                                          */

}

/* ------------------------------------------------------------------------------------------------------------*/
void        user_init_menu()            /* User defined menus here                                          */
{
 /*..... Place user defined menu initializations here.                                                       */

 main_menu = defpup(" Sample Main Menu      %t ");
 addtopup(main_menu," Place menu choices here %x100 ");
 addtopup(main_menu," Quit Program          %x999 ");
}

/* ------------------------------------------------------------------------------------------------------------*/
void        user_init_cursor()          /* User defined cursor init                                         */
{
 /*..... Place user defined cursor initializations here.                                                     */

}

/* ------------------------------------------------------------------------------------------------------------*/
void        user_init_overlay()         /* User defined overlay init                                        */
{
 /*..... Place user defined overlay initializations here.                                                    */

}

/* ------------------------------------------------------------------------------------------------------------*/
void        user_init_main()            /* User defined main initializations                                */
{
 /*..... Place user defined initializations here.                                                            */
 /*..... This is called after all panel and actuator setup initializations.                                  */

}

/* ------------------------------------------------------------------------------------------------------------*/
void        user_process_queue          /* User defined queue functions                                     */
(
 short      TOKEN,                      /*  Graphics event Q device token                                   */
 short      TOKvalue                    /*  Graphics event Q token value                                    */
)
{
 /*..... Place user defined queue processing here.                                                           */
 /*..... All queued tokens will be passed to this function after they are                                    */
 /*.....  processed by the Panel ToolBox functions.  They may be used by                                    */
 /*.....  the User's program to specify additional actions, etc.                                            */

}
```

```
/* ------------------------------------------------------------------------------------------------------------*/
void            user_process_menu()             /* User defined menu processor                              */
{
 long           choice;

 choice = dopup(main_menu);

 switch ( choice ) {

  /*----- Include other menu selection processing here.                                                       */

  case MENU_QUIT:                               /* exit the program                                         */
   user_exit();
   break;

  default:
   break;
 }

}

/* ------------------------------------------------------------------------------------------------------------*/
void            user_display()                  /* All user calc & drawing functions                        */
{
 /*..... Place user defined calculations and display control here.                                            */
 /*..... This is called during each drawing loop after control panel                                          */
 /*.....  processing is completed. */

}

/* ------------------------------------------------------------------------------------------------------------*/
void            user_exit()                     /* Clean up and exit the program                            */
{

 /*..... Place user defined exit procedures here.                                                             */

 panelExit();                                   /* Clear and close all Panel windows                        */

}


/*****************************************************************************
*  EOF: User_Panel_fn.c       { lines:   85 }                                  *
 *****************************************************************************/
```

```
/*****************************************************************************
*  File:     User_Panel.h                    User modifiable constants and decls   *
*  Version:  1.0                                                                   *
*  date:     90/11/17                                                              *
*  Author:   Richard M. Prevatt                                                    *
*            David M. King                                                         *
*  ---------------------------------------------------------------------------    *
*  Notes:                                                                          *
*            90/08/13  Created.                                                    *
*  ---------------------------------------------------------------------------    *
*  This file contains header information as generated by PanelDesigner.           *
*  It is used in conjunction with User_Panel.c and User_Panel_fn.c                *
*                                                                                 *
*  The actual name of this and the related files was derived from the             *
*  name of the current workspace when it was produced by PanelDesigner.          *
*  { Substitute the actual name for 'User_Panel' in these instructions. }         *
*                                                                                 *
*  If a file by that name already existed, the PanelDesigner saved the            *
*  the original version in a backup file as follows:                              *
*    User_Panel.h  -->  User_Panel.h.bak                                          *
*                                                                                 *
*  Compile as follows:                                                            *
*                                                                                 *
*    cc -o user_name User_Panel.c User_Panel_fn.c /nps_path/lib/ npspanel.a        *
*     -I/nps_path/include -O2 -align16 -G 0 -lc_s -lgl_s -lfm  -lm                *
*                                                                                 *
*            /nps_path must be defined as the proper path to the NPS Panel ToolBox *
*                                                                                 *
*            /nps_path = /n/gravy1/work/zyda/npspanel in the current release      *
*                                                                                 *
*  The resulting file 'user_name' may be executed.                                *
*****************************************************************************/

#ifndef     user_PANEL
#define     user_PANEL

/*  ---------------------------------------------------------------------------*/
/*  Global constants */

#define     MAX_PANELS          1       /* Max number of panels defined       */
#define     MAX_ACTUATORS       4       /* Max number of actuators defined    */

#define     FONT_FACTOR         12.0    /* Scale factor for font manager      */

#define     MENU_QUIT           999     /* Menu selections are define here    */


/*  ---------------------------------------------------------------------------*/
/*  Global reference variables */

EXTERN                                  /* User control panel reference       */
Panel       *Control_Panel[MAX_PANELS];

EXTERN                                  /* Actuator reference array           */
Actuator    *Control[MAX_PANELS][MAX_ACTUATORS];
```

```
EXTERN
long        main_menu;              /* Main menu reference                  */

/* ------------------------------------------------------------------------*/
/* Function prototypes for user modifiable functions */

void        user_init_main();       /* User define main initializations     */
void        user_init_cursor();     /* User defined cursor init             */
void        user_init_overlay();    /* User defined overlay init            */
void        user_init_queue();      /* User defined queue init              */
void        user_init_menu();       /* User defined menu init               */
void        user_process_queue      /* User defined queue functions         */
(
  short,
  short
);
void        user_process_menu();    /* User defined menu processor          */
void        user_display();         /* All user calc & drawing functions    */
void        user_exit();            /* Clean up and exit the program        */


#endif      user_PANEL

/*****************************************************************************
* EOF:      User_Panel.h      { lines:  84 }                                  *
 *****************************************************************************/
```

APPENDIX E

NPS PANEL DESIGNER AND TOOLBOX SAMPLE INTERMEDIATE FILE

```
Panel_Designer_File

Panel Box_Dial
/C This is an example of an optional permanent comment line C/
/* panel x, y, w, h */                          10 10 980 700
/* auto_align, grid_on, grid_size */            0 0 25.0
/* visible, selectable, fixed, popable */       1 1 0 0
/* border, screen_relative, zbuffer */          1 1 0
/* wl, wr, wb, wt, wn, wf */                    0.0 980.0 0.0 700.0 0.0 0.0
/* scale_factor, color_table */                 1.0 0

  Actuator BOX
  /* type, group_id, key_equivalent */          10 -41 0
  /* active, visible, selectable*/              0 1 1
  /* x, y, w, h, bw */                          469.5 208.5 85.0 25.0 0.0
  /* color_table */                             0
  /* l_location, label, label_font */           -13 "Box" 12.0
  /* lx, ly, lw, lh, lbx, lby */                24.2 1.7 36.6 21.6 4.8 5.8
  /* v_location, value_fmt, value_font, val */  0 "%-+#4.2f" 12.0 0.0
  /* initval, minval, maxval */                 0.0 0.0 1.0
  /* vx, vy, vw, vh, vbx, vby */                21.2 -27.6 42.6 21.6 4.8 5.8
  /* line_width, frgnd_clr, bkgnd_clr */        2 0 -1

  Actuator DIAL
  /* type, group_id, key_equivalent */          40 -23 0
  /* active, visible, selectable*/              0 1 1
  /* x, y, w, h, bw */                          671.5 402.5 75.0 75.0 2.0
  /* color_table */                             0
  /* l_location, label, label_font */           2 "Dial" 10.0
  /* lx, ly, lw, lh, lbx, lby */                22.5 -24.0 30.0 18.0 4.0 5.0
  /* v_location, value_fmt, value_font, val */  0 "%-+#4.2f" 12.0 0.0
  /* initval, minval, maxval */                 0.0 0.0 1.0
  /* vx, vy, vw, vh, vbx, vby */                16.2 -27.6 42.6 21.6 4.8 5.8
  /* mode, shape, r, major_tics, minor_tics */  2 1 33.8 4 0
  /* tl, tw, ml, mw */                          11.8 2.7 32.1 2.7
  /* theta, winds, finefactor */                0.0 1.0 0.1

Panel_End

Custom_Colors

File_End
```

LIST OF REFERENCES

Barth, Paul S. "An Object-Oriented Approach to Graphical Interfaces." *acm Transactions on Graphics*. Vol 5 No 2, April 1986, pp. 142-172.

Brown, C. M., D. B. Brown, H. V. Burkleo, J. E. Mangelsdorf, R. A. Olsen, and R. D. Perkins. *Human Factors Engineering Standards for Information Processing Systems* (LMSC-D877141). Lockheed Missiles and Space Company, Sunnyvale, CA, 1983.

Brown, Judith R. and Steve Cunningham. *Programming the User Interface*. John Wiley & Sons, Inc., New York, 1989.

Danchak, M. M. "Alphanumeric Displays for the Man-Process Interface. Advances in Instrumentation." ISA Conference, Niagara Falls, New York, October 1977, pp. 197-213.

Engel, S. E. and R. E. Granda. *Guidelines for Man/Display Interfaces* (Technical Report TR 00.2720). IBM, Poughkeepsie, NY, 1975.

Fischer, Gerhard. "Human-Computer Interaction Software: Lessons Learned, Challenges Ahead." *IEEE Software*, January 1989, pp. 44-52.

Foley, James. "Guest Editor's Introduction: Special Issue on User Interface Software." *acm Transactions on Graphics*. Vol. 5 No. 4, October 1986, pp. 279-282.

Gailtz, W. O. *Human Factors in Office Automation*. Life Office Management Association, Atlanta, GA, 1980.

Goodwin, Mark. *User Interfaces in C*. Management Information Source, Inc., 1989.

Haeberli, Paul E. "a data-flow manager for interactive graphics." *Iris Universe*, Fall 1987, pp. 3-5.

Hill, Ralph D. "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction--The Sassafras UIMS." *acm Transactions on Graphics*. Vol 5 No 3, October 1986, pp. 179-210.

Jacob, Robert J. K. "A Specification Language for Direct-Manipulation User Interfaces." *acm Transactions on Graphics*. Vol 5 No 4, October 1986, pp. 283-317.

Limanowski, J. J. "On-line documentation systems: History and issues. *Proceedings of the Human Factors Society 27th Annual Meeting* Human Factors Society, Santa Monica, CA,1983, pp. 1027-1030.

MIL-STD-1472D, Revised 14 March 1989. *Military Standard: Human Engineering Design Criteria for Military Systems, Equipment and Facilities*. Department of Defense, Washington, DC, 1983, pp. 247-278.

Miller, G. A. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity For Processing Information." *Psychological Review*, Vol. 63, No. 2, 1956, pp 81-97.

NASA (National Aeronautics and Space Administration). *Spacelab Experiment Computer Application Software (ECAS) Display Design and Command Usage Guidelines.* (Report MSFC-PROC-711). George C. Marshall Space Flight Center. 1979.

Olsen, Dan R. Jr. "MIKE: The Menu Interaction Kontrol Environment." *acm Transactions on Graphics.* Vol 5 No 4, October 1986, pp. 318-344.

Parsons, H. M. "The scope of human factors in computer-based data processing systems." *Human Factors*, Vol. 12, 1970, 165-175.

Pfaff, G. (editor) *User Interface Management Systems.* Springer-Verlag, New York, 1985.

Phillips, R. J. "An experimental investigation of layer tints for relief maps in school atlases." *Ergonomics*, Vol. 25, 1982, pp. 1143-1154.

Ramsey, H. R. and M. E. Atwood. *Human Factors in Computer Systems: A Review of the Literature* (Technical Report SAI-79-111-DEN). Science Applications, Inc., Englewood. CO, (NTIS No. AD A075 679), 1979.

Reid, Pete, "Work Station Design, Activities and Display Techniques", *Fundamentals of Human-Computer Interaction*, Andrew Monk, editor. Academic Press, 1985, pp. 107-126.

Sidorsky, R. C., R. N. Parrish, J. L. Gates, and S. J. Munger. *Design guidelines for user transactions with battlefield automated systems: Prototype for a handbook* (ARI Research Product 84-08). US Army Research Institute, Alexandria, VA, (NTIS No. AD A513 231), 1984.

Smith, Sidney L. "Man-computer information transfer." *Electronic Information Display Systems*, J. H. Howard (Ed.), pp. 284-299. Spartan Books, Washington, DC, 1963.

Smith, Sidney L. and Jane N. Mosier. "The user interface to computer-based information systems: A survey of current software design practice." *Behaviour and Information Technology*, Vol. 3, 1984, 195-203.

Smith, Sidney L. and Jane N. Mosier. *Guidelines for Designing User Interface Software.* MITRE, Bedford, Massachusetts, ESD-TR-86-278, Electronic Systems Division, AFSC, 1986.

Schneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Reading: Addison-Wesley Publishing Company, 1987.

Thimbleby, Harold. "User Interface Design: Generative User Engineering Principles." *Fundamentals of Human-Computer Interaction.* Andrew Monk, editor. Academic Press, 1985, pp. 165-180.

Wilson, Stephen H. "the layered user interface." *Iris Universe,* fall 1987, pp. 9-11.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Michael J. Zyda Naval Postgraduate School Code CS, Department of Computer Science Monterey, CA 93943-5100	7
4.	Dr. H. Loomis Naval Postgraduate School Code EC, Department of Electrical Engineering Monterey, CA 93943-5100	1
5.	Lieutenant David M. King 1151 Aquidneck Avenue Suite 402 Middletown, RI 02840	1
6.	Lieutenant Commander Richard M. Prevatt 6908 Conservation Drive Springfield, VA 22153	1
7.	David Pratt Naval Postgraduate School Code CS, Department of Computer Science Monterey, CA 93943-5100	1