Tajo Functional Specification

Version 6.0
October 1980

This document is for Xerox internal use only

XEROX
OFFICE PRODUCTS DIVISION
SYSTEMS DEVELOPMENT DEPARTMENT
El Segundo / Palo Alto California

Table of Contents

1.0 INTRODUCTION AND SCOPE

This document describes **Tajo**, a collection of interfaces that provide the framework and runtime system for the Mesa Development Environment.

Throughout this document the term *user* describes people who interact with Tajo via the mouse, keyset and keyboard. The term *client* is used to describe programs and programmers that use the interfaces described herein.

Terms that occur in the glossary of this document are bold the first time they are used, e.g., **"Tajo"** above. All Mesa code and names of variables or types that are defined in the definitions modules appear in this sans serif font (Mesa reserved words appear in SMALL CAPITALS). Names that are part of the interface are in boldface when used in the text. This small font is used for fine points.

1.1 Introduction to Tajo

Tajo is designed to facilitate the implementation and execution of a wide range of software development programs. These programs are referred to as **Tools**. In the strict sense, any Mesa program that will execute in Tajo is a Tool. However, considerably more semantics are applied to the term Tool in this document.

This document describes the interfaces provided by Tajo that are utilized in building Tools. It is intended primarily for writers (programmers) of Tools. Advice and hints about how a Tool can best use the facilities provided is also given. The documentation (not the definitions modules) is considered to be the final word on interfaces implemented by Tajo. It is expected that clients of Tajo will write code from this specification.

This document is a reference manual, not a tutorial. Sec. 3 contains a motivation of why the separate facilities of Tajo exist and explains their expected use. Sec. 4 describes in more detail the design and philosophy of Tajo and offers some stylized patterns of use that have evolved during the development of Tajo. The detailed descriptions of the interfaces are contained in Sec. 5. Section 6 describes some AltoMesa and Pilot operational considerations. Example programs are listed in the appendices. The recommended easiest way for a client to become proficient with Tajo is to start with one of these Tools and to modify it into the Tool desired.

1.2 The Tajo User Illusion

The following is a brief description of the Tajo user illusion. A complete description is found in the *Guide For Tool Users.*

From the user's point of view Tajo provides uniform and consistent interaction modes for both inter-Tool and intra-Tool actions. This is accomplished by providing a user illusion that consists of standard mechanisms and procedures for interacting with Tools. Central to Tajo is the notion of breaking the user's interactions down into a reasonably small number of atomic actions when

specifying parameters and commands. Long sequences and context dependent interactions are discouraged. The goal is to make similar actions, in different contexts, have predictable results.

The user interface for Tools provides the unifying framework for Tajo. Tools utilize display **windows** to present information to the user. They all receive input and commands from the user in the same using the same mechanisms. The user is encouraged to have an arbitrary number of Tool windows on the display screen. The windows may overlap each other, some obscuring portions of others. The user directs his actions to the Tool whose window contains the cursor. The primary command invocation mechanisms are via **menus** and command items contained in **forms**. In addition, facilities are provided for running new Tools and manipulating windows on the display screen.

2.0 REFERENCES

Functional Specification. Vista: SDD Display Window Package October, 1980.

Mesa Language Manual Version 5.0 April, 1979.

Mesa System Documentation Version 6.0 October, 1980.

OIS Software Tools Environment Functional Specification January, 1977.

The above document discusses the whole of the Development Environment and several individual Tools from the viewpoints of both a human user and a programmer client. Where the present document parallels the above document, the above document is to be taken as superceded.

The Pilot Programmer's Manual Version 5.0 October, 1980

Tajo: Guide For Tool Users Version 6.0 October, 1980

SDD Software Development Procedures and Standards April, 1977.

3.0 CONCEPTS AND RELIGION

The following sections describe the fundamental facilities that are the building blocks for Tajo and implementation of Tools. These facilties implement and make concrete the fundamental ideas and philosophy of Tajo. These facilities are:

Windows and subwindows
User input
Menus
Selections
Forms
Text

3.1 Windows

The concept of the window provides the mechanism by which Tajo insulates functions from one another on the display. A window is operated upon without regard for the its position on the physical display screen.

A client commonly creates a single *tool* window. Overlapping between a tool window and other tool windows is ignored by the client. Usually, a client wishes to consider that the tool window is divided into a number of rectangular areas, **subwindows**, that are to be separately acted upon. Subwindows are used for actually displaying client data: textual information, graphical constructions or pictures. Tajo provides appropriate primitive procedures to facilitate each of these. Although "window" is usually a generic term in this document, it occasionally acquires from the surrounding context the restriction that it be tool window.

A digression is in order here to describe the relationship between Tajo and the software package, Vista, that it uses to implement windows. Vista implements a general tree of windows. Vista has no notion of depth dependent window specialization within the tree (except for the top level). Thus, Vista allows a window to be moved from any depth in the tree to any other depth.

Tajo does not preclude clients from utilizing all of Vista's generality, but it does not use all of the generality itself. Tajo defines five types of windows corresponding to the depth of the windows in the tree and applies specific semantics to each type. The topmost window in the tree is called the *root* window. It is the size of the bitmap, and is used to define the dimensions of what the user can see. The next level are tool windows. The third level windows are called *clipping* windows and are used to define the "inside" or clipping box of tool windows for display purposes. Usually clients shouldn't have to deal with clipping windows. The fourth level are subwindows. All windows of depth greater than four are treated by Tajo as *other* windows, with no Tajo supplied specialization.

3.2 User Input

Given that a user may be simultaneously interacting with numerous Tools, Tajo must provide a mechanism for specifying to which Tool (or window) any particular user action (mouse, keyboard, or keyset activity) is directed. The requirement to direct user actions to various windows has given rise to a **notification mechanism.**

3.2.1 Notification

This mechanism is designed to be flexible and have minimal overhead for the processing of user input actions. The basic scheme is that a Tool may *install* a number of procedures, one for each user action, to be called (notified) when a change in the user input state occurs. A Tool is allowed to get control immediately when the state change occurs, or at a later time when other user initiated processing has completed. The rest of this subsection can be skipped during an initial reading.

The Notification mechanism is basically split into two distinct processes: 1) a high priority process, called the **Interrupt Level**, for queuing user actions, and 2) a normal priority process, called the **Notifier** (or **Processing Level**), for the actual processing of user actions.

The Interrupt Level gets control of the cpu every vertical retrace. This is around 50 times per second, depending on the processor type. This level watches for changes in the hardware user state, i.e., mouse movements or key station (*paddle* on the keyset, *button* on the mouse, or *key* on the keyboard) depressions or releases. A Tool may have Stimulus Notification Routines (SNR's) that get control at this level.

> Warning: The Interrupt level of the notification mechanism is a high priority process and as such has potential preemption difficulties (in particular it cannot cause code swapping while it is executing on the Alto). This has serious implications and if you plan to use this facility. Please consult with Tajo implementors.

The Notifier's primary function is to call Tool supplied Processing Notification Routines (PNR's). The Interrupt Level communicates with the Notifier via a queue. Basically, the Notifier dequeues the head item from the queue and the appropriate PNR is called. The secondary Notifier function is to process the queue of periodic notifiers. This is a queue of procedures that wish to run on a periodic basis, but which do not want to execute unless the Notifier would be otherwise unoccupied. If it is idle the Notifier WAITs on a condition variable so that other processes (e.g communications, etc.) may run.

> [Note: When PNRs or periodic notifiers are executing they are extensions of the Notifier (i.e. on the same call chain). This means that notification processing is suspended or "backs up" until control is returned from the PNR. If you intend to do some serious computing you should FORK a process and let the Notifier get back to its task.]

3.2.2 TypeIn

The TypeIn facility lets the client supply a procedure that will be called whenever a character is typed. It is important to note that the client gets called upon input of a character instead of calling a GetChar procedure for one. This has a major impact upon the control structure of Tools, requiring more explicit program state data. The user TypeIn facilities are built using the above described notification facilities. TypeIn allows the client to essentially be free of any concern for "how it is done".

3.3 Menus

A menu is a set of options and commands associated with a window. It is possible to have multiple menus on a single window. When the menus associated with a subwindow are displayed the menus associated with its tool window are also displayed.

A menu is implemented as an array of items. Each menu item is a pair, a keyword and a **menu command routine** (a procedure to be called when the keyword is chosen by the user). A new menu instance is created each time a menu object is associated with a window. When the same menu appears in multiple windows, the menu object itself is shared.

3.4 Selections

An example of a **selection** is a text string or a graphic icon that the user has caused to become highlighted in some way. Many commands operate on the **current selection.** Tajo provides a mechanism by which windows pass around "ownership" of the current selection. A Tool may operate upon the current selection even though it is not in its own window.

Some commands may require more than one parameter. These commands usually gather their parameters through the use of forms (see Sec 3.5), instead of using the selection mechanism.

3.5 Forms

A Tool often needs to collect more than one parameter from the user, but the collection of multiple parameters using only menus and the current selection is difficult to program, and tends to create a clumsy (and often inconsistent) user interface. In addition, such a Tool often wishes to display both the current parameter settings and internal program state data. It also wants to allow modification of the parameters.

The Tajo forms package frees the Tool writer from the concerns of parameter collection, display and modification. The package shields the Tool writer from the display management tasks of making selections, displaying characters, wrapping long lines and scrolling. A form also provides a uniform and consistent way for the user to interact with many different Tools. A *mode* problem arises whenever commands require multiple arguments. In most systems, each such command enters a special mode in which the user can only perform actions that are pertinent to the collection of the current command's parameters; the parameters themselves are often collected in a specific order that does not allow modification of any parameter except the one currently being collected. In Tajo, this is unacceptable because the system should not preempt the user. Thus, a form allows the user to enter or modify any or all parameters in any order prior to command execution.

The forms package is implemented so that the Tool writer has access to various levels at which he can be involved in the display and alteration of the fields in the form. However, the interface is primarily aimed at two distinct clients: the Tool writer who only wants minimal control, and the Tool writer who wishes absolute control over parameter display and alteration. Some facilities are provided for clients that wish to be somewhere in the middle of these two positions.

3.6 Text

All text displayed by Tajo is managed by a common set of display and user input facilities. This allows the user to have one consistent way of selecting and modifying text. This management is provided by two packages, a low level one that does not know about the display but which provides the primitive operations on the concrete representation of the text (i.e. disk operations for text that is coming from files, string operations for text that is stored only in primary memory, etc), and a high level package that handles the displaying of the text and interactions with the user trying to manipulate the text.

3.7 Cursors

The cursor management routines are a small set of definitions and procedures which provide, in essence, a *virtual* interface to the hardware cursor.

These routines provide for the setting of the 256-bit pattern which is displayed by the hardware as the human-visible cursor. They also provide for the specification of the *hot spot*, the position within the 16x16 bit cursor that is meant to indicate the screen position pointed to by the mouse. Since hardware implements the 16x16 bit cursor shape, but does not implement a hot spot, the implementing routines themselves contain what might be considered the *hardware hot spot* coordinates.

3.8 Scrollbars

In general, Tajo has no knowledge of the contents of a window or subwindow so the actual scrolling operations (i.e. moving the bits) are the responsibility of the client. Tajo does provide a standard user illusion and mechanism for the user to invoke and cause the client to perform scrolling operations.

3.9 Librarian Interface

The **Librarian** Interface is the component in Tajo whose primary task is to provide a procedural interface to a physically remote function. The Librarian Interface is the Tool writer's access mechanism to the contents of objects, **Libjects**, stored in the **Librarian Data Base** and under the control of the **Librarian Service** Tool. It is intended to be the *only* method of access to these data.

The primary purpose of the Librarian Interface is to provide a uniform procedural interface to the Librarian Service. The interface serves to define and enforce both the syntax and semantics of the basic request/response activity. It further insulates its users from the *how's* and *where's* of Librarian functions as well as insulating the Librarian Service from malformed requests.

Most Libjects are used as access locks for Mesa source files to prevent simultaneous editing. If these files are managed by Tools that use the Librarian Interface, then the classical simultaneous conflicting edits problem is avoided. Each time the Libject lock is acquired *(checkout)* and released *(checkin)* a new version of the Libject is created. All versions of a Libject are kept **forever**, allowing the history of a project to be traced from its conception. The type of historical data maintained includes the who, why, what, where and when of lock operations.

3.9.1 Property Lists

The Librarian Interface needs a standard way of discussing the various elements that make up a Libject version. This is made difficult by the fact that neither the Librarian Interface nor Librarian Service *know anything* about the items that it stores, and therefore cannot, in general, know their TYPEs or (in particular) their SIZEs. The resolution of this difficulty lies in the creation of the **PropertyList**.

- A **PropertyList** is an array of **PropertyPairs**. Each **PropertyPair** defines an item within a Libject-version.

- A **PropertyPair** consists of a **PropertyNumber** and a **PropertyValue**, the former serving to name the particular item-type, and the latter serving to contain the information.

To allow the Librarian Interface to manipulate **PropertyPairs** without knowing about each type of property, we assign **PropertyNumbers** in a manner which supplies some information about the **PropertyValue** from an examination of the **PropertyNumber.** In particular our scheme allows a determination of both the **SIZE** of the **PropertyValue** and whether the **PropertyValue** is a **LibjectID** (a Librarian Interface assigned identifier) from examination of the **PropertyNumber.**

3.10 Basic precepts (The Commandments)

The above facilities were designed with some specific goals in mind. These goals are a result of a philosophy that can be summed up in the following "religious commandments".

Clients shall not preempt the user. (Swinehart's Law)

This is Tajo's basic tenet. The user should never be forced by the clients into a situation where the only thing that he can do is interact with only one Tool. Even stronger, the client should try to avoid falling into a particular "mode" when interacting with the user, i.e. the Tool should try to avoid imposing unnecessary restrictions on the permitted sequencing of user actions.

Don't call us, we'll call you. (Hollywood's Law)

A client should never sieze control of the processor while getting user input, which is exactly what tends to happen when the client wants to use the "get a command from the user and execute it" model of operation. Instead, the Tool should arrange for Tajo to notify it when the user wishes to communicate some event to the Tool.

The user owns the window layout.

Although it is possible for the client to re-arrange the window tree and the positions and sizes of the windows on the display, this is discouraged. Each user has particular and differing tastes in the way that he wishes to lay out windows on the display, and it is not the client's role to overide the user's decisions. In particular, clients should avoid having windows jump up and down trying to capture the user's attention. If the user has put a window off to the side, then he does not want to be bothered by it.

The display belongs to the Notifier.

Although an attempt has been made to accommodate clients that want to use multiple processes, the display interactions are sufficiently fragile (especially in the presence of the destruction of windows) that the display is considered to belong to the Notifier. If some *background* process (here all non-Notifier processes are referred to as background processes) wants to do something that could affect the display, it must make sure that the routines it calls are explicitly declared to

be callable from a background process.

4.0 SOFTWARE DESIGN OVERVIEW

This section provides an overview of the interactions between Tajo's major software components. Pieces of Tajo are delineated and the some of the interdependencies are described. Hints are offered on how the client programmer might best use Tajo facilities.

Clients who are interested in writing only a simple Tool and who have some idea about how Tajo is structured might consider skimming the following sections up to the section entitled *Tool Interface.* This section gives an overview of the top layer of software that a client deals with when writing a simple tool.

4.0.1 Tajo's Components

The most basic portion of Tajo deals with user input devices, specifically the keyboard, keyset and mouse, and the display output devices, specifically the cursor and bitmap. This portion can be changed by the client without badly affecting the rest of Tajo provided that the changes are made by replacing the SNRs and PNRs. The interfaces of interest are Keys, UserInput and UserTerminal for input, Cursor and Vista's Window for output. Further routines that Tajo uses to augment the Window and UserInput interfaces are provided by the Context and ToolWindow interfaces.

The next level of output complexity is the simple display of text. The interfaces of interest are ToolFont, Window and WindowFont.

Continuing along the axis involving text on the display, the next level is involved with more complicated display of text. The interfaces of interest are TextDisplay and TextSource. TextDisplay is a private interface but some clients, particularly those contemplating creating their own subwindow types, might need to use it. If you find this is necessary you should consult a Tajo implementor. The interfaces Caret, Scrollbar and Selection are concerned with text marking and user input.

These lower level functions are used to implement Tajo supplied subwindow types described by the interfaces: FileSW, FormSW, MsgSW, StringSW, TextSW and TTYSW. The Menu interface provides simple command invocation and is required by several of the subwindow types.

The combining of subwindows within a window to make a Tool is facilitated by the Tool interface. The FileWindow and TTY interfaces enable the creation of Tools that each contain one subwindow of a predefined type. The same caveat applies to FileWindow that applies to TextDisplay.

To one side are STRING and formatting utilities. These cannot be discarded without affecting some of the higher levels of Tajo such as the subwindow types. The interfaces of interest are Format, HeapString and Put.

There are several interfaces dedicated to interacting with the surrounding runtime environment, and to informing clients of changes in that environment. The interfaces of interest are CmFile, Event, Profile and TajoMisc.

Implemented as a strict add-on are the routines used to converse with the remote Librarian Service: they are provided by Librarian.

4.0.2 Tajo Naming Conventions

Although there are many exceptions to these rules, the following are the general naming conventions practiced in Tajo.

Procedure types have the suffix **ProcType**. These types always use keyword notation to name their arguments, and when there is more than one return value they are also named. The same convention for naming arguments in the procedure types is used for naming arguments in the interface procedures. In general, the argument is called: **window** if it is a general window, **sw** if it is a Tajo subwindow, **ss** if it is a **String.SubString**, and **index** if it is a CARDINAL identifying a position in some descriptor. There is extensive use of defaults.

The primary object supported by an interface is a TYPE named **Object**. Secondary objects have suffix **Object**. A POINTER TO **Object** has suffix **Handle**.

Most of the interfaces have one signal called **Error** used to note exceptional conditions (usually client errors or failures of the runtime environment). This signal has an argument called **code** which is an **ErrorCode** defined in the same interface as the signal. The signal cannot be RESUMEd except when this is explicitly permitted.

Interfaces that implement subwindow types have procedures named **Adjust**, **Create**, **Destroy**, **IsIt**, **Sleep** and **Wakeup**. However, note that **Create** and **Destroy** do not actually create and destroy subwindows; rather, they add or remove specialization to existing Tajo subwindows.

Private definitions modules usually have the substring *Ops* appended to the name of the associated Public definitions module. For implementing modules the suffixes are *Impl* or *sA*, *sB*, *sC*, etc. Bound groups of implementing modules have suffix *s*.

An enumerated type is often referred to as being "open ended". This means that it is declared to be a MACHINE DEPENDENT enumerated type, and that room has been left in the enumeration for future expansion or to allow Tajo to generate unique, un-named elements of the type at runtime.

4.1 Subwindows

The subwindow is the fundamental object from both the user's and the client's points of view. This should not be surprising since user actions are directed at specific subwindows; as a consequence subwindow handles are constantly passed back and forth between clients and Tajo. This leads to the practice by both Tajo and clients of associating data about the user's current context with subwindows.

Subwindows come in various degrees of complexity. A raw subwindow is created by calling **ToolWindow.CreateSubwindow**, which results in a **Window.Handle** and additional associated data used by Tajo. These data include, but are not limited to, the subwindow's PNRs. Such a subwindow is not very interesting for it does nothing but display white bits, and either ignores or gives away all

user actions directed to it. Tajo supplies several interfaces that support particular styles of subwindow enhancement.

The next level of complexity is realized by replacing the subwindow's display procedure by one that shows the state of some associated client data structure.

The higher levels of complexity affect the user input and display output facilities of the subwindow (such as the **PNR**s and **StringIn** and **StringOut** routines defined in the `UserInput` interface), usually by replacing them with more powerful ones. It is not uncommon for different subwindow types to share common input routines (a good example of this is the **Menu.PNR** which is often the PNR for the yellow mouse button), or to share the marking and resolution routines (such as those involved in the selection of text) provided by lower levels of Tajo.

4.2 System Supplied Subwindow Types

Tajo provides subwindow types which display well defined client data structures in a particular stylized layout. Some of these types are built upon the others; however, it is not possible to arbitrarily mix these types in a single subwindow (for instance a subwindow cannot have the attributes of both Form and File subwindow types).

The process of calling a system supplied subwindow type **Create** routine is usually termed "creation" of a system supplied subwindow type. This is a misnomer, because it is actually a differentiation process; the client must have already created the subwindow to be operated upon by a call to **ToolWindow.Create**.

4.2.1 Form Subwindows

Clients of the **FormSW** interface need to supply a procedure that completely defines the form that is to be presented to the user. Much of the detail of the form layout and individual item options can be defaulted, thus permitting the direct specification of only those things that are important to the client. Much of the work of defining the form is accomplished by calls to **FormSW.*Item** procedures where * can be one of {**Boolean**, **Command**, **Enumerated**, **Label**, **LongNumber**, **Number**, **String**, **TagOnly**}. In addition, the client may supply notification routines that get called when the user alters the form items.

4.2.2 Text Subwindows

The `TextSW` interface presents a set of procedures that define a top level of uniform text displaying and editing capabilities that are used by higher level subwindow types. A subwindow of this type allows the client to present a menu to the user that allows him to search for text strings, normalize the insertion point to the top of the display region, normalize the selection to the top of the display region, jump to the n^{th} character in the text, split the subwindow into separately scrollable display regions and change the line break mode.

4.2.2.1 Text Sources

When a TextSW is created it is passed a **TextSource.Handle**. This handle contains a set of procedures that are used to manipulate the backing data structure, the "source", that holds the characters displayed by the TextSW. Thus, creating a TextSW with a client defined source, the data representation can be of any suitable form.

4.2.2.2 Text Subwindow Types

Two system supplied subwindow types are built directly on top of the TextSW mechanism. One of these is defined by the `FileSW` interface and uses a file as the source of its characters. This interface has procedures that do file specific operations; other operations may be found in the `TextSW` interface. The `StringSW` interface supports a subwindow type that uses a STRING as the source.

4.2.3 Message Subwindows

The `MsgSW` interface defines a set of procedures for posting messages to the user. The client may direct that in addition to writing to a MsgSW he wishes the output to go to another subwindow as well. This is useful for logging error messages. MsgSWs are built on top of StringSWs. So, if the procedure that you need isn't in the `MsgSW` interface then it might be in the `StringSW` or `TextSW` interface.

4.2.4 TeleType Subwindows

In Tajo, client programs are notified of user keystrokes; this is in contrast to teletype environments in which the client polls some keyboard handler for type-in. Many programmers are either used to or have code that is structured to use a teletype like interface. To accommodate this style of user interaction there is a system supplied subwindow type defined by the `TTYSW` interface. TTYSWs are built on top of FileSWs.

4.3 Client Subwindow Types

There is considerable flexibility built into the system supplied subwindows. However, radical changes in the style of the layout of the subwindow, changes in the definition of the client data structures or the alteration of the details of the user input interactions can only be achieved by the writing of new code. This results in client implemented subwindow types. The implementors of Tajo encourage clients that find it necessary to implement their own subwindow types to do so in the same manner in which the system supplied types are constructed; namely to have a stand alone package, complete with a definitions module and documentation, that can be loaded as a single, possibly bound, .bcd file into Tajo.

There are two possible approaches to implementing a new subwindow type. The first is to build upon an existing subwindow type without changing the implementation code for that type. This can be done only by strict *augmentation* or by the process of *interposition.* By augmentation we mean

adding totally new facilities to a subwindow type without conflicting or modifying existing facilities. This is difficult to do with the Tajo supplied types because each one of them has a full complement of user input and output features. The more common approach is interposition; the idea is to replace one or more of the procedures provided or used by an existing subwindow type with client procedures, either by changing the binding path or by replacing procedure variables. This allows the client to detect the occurrence of the event that the replaced procedure is associated with and to do pre- or post-processing.

An example of interposition that occurs in Tajo is an empty String subwindow's recognition of the escape character as a command to notice the current text in the subwindow. The implementation of this is to replace the keyboard PNR of the String subwindow with another PNR which watches all of the keyboard input go by. When an escape is typed by the user and sent to the PNR, instead of sending it on to the regular PNR, the interposed PNR makes the contents of the String subwindow the current selection and then looks at the text of the current selection.

[Fine point: A deliberate attempt has been made to allow interposition of any system PNR. This is accomplished by setting up the PNR only when the subwindow is created. If it was done at any other time it might overwrite an interposing PNR. In-other-words, if the subwindow type wants a PNR to behave differently in a time-dependent manner, it should set flags rather than changing the PNR. No such claim is made for the interposition of a subwindow display procedure.]

4.3.1 Implementing A Package

A subwindow type that would be useful to others should implement the **Adjust**, **Sleep** and **Wakeup** procedures (although they need not have those names, see **Tool.RegisterSWType**). Good citizenship encourages the implementation of the **Destroy** and **IsIt** procedures, while common sense dictates that there be a **Create** procedure.

A common problem that arises when implementing a subwindow type is where to keep the state data for the subwindow. System defined subwindow implementations associate the data directly with the **Window.Handle** using *contexts*, defined by the interface **Context**. The next two paragraphes can be skipped on the first reading and should only be read after studying the Context interface.

The typical scenario for a client using contexts to keep his state data in has three phases plus a start up transient. The transient occurs when the subwindow type implementor is first STARTed. At that time the implementor gets a unique *context type* from Tajo. Then when a subwindow is differentiated by a client calling the implementor's **Create** procedure (the initialization phase), the implementor creates a context with the previously obtained unique type on that subwindow. The **data** argument of the context creation call is a pointer to the subwindow's state data.

Whenever Tajo calls the implementor to operate upon the subwindow (the steady state phase), a subwindow handle with which the implementor can get the state data is provided. Eventually, the finalization phase is reached when either the client calls the implementor's **Destroy** procedure or Tajo decides to destroy the subwindow. When Tajo destroys the subwindow, it causes all the contexts associated with the subwindow to be destroyed. The implementor's **Context.DestroyProcType**, which should free the state data, is called. The implementor's **Destroy** procedure should call **Context.Destroy**.

4.4 Tool Interface

The Tool interface is designed to make the writing of Tools with a standard user interface as easy as possible. It allows the client to avoid understanding many of the low level facilities provided by Tajo in exchange for some loss in flexibility.

In windows created using the Tool interface the user has the ability to dynamically move the horizontal boundaries between subwindows. The interface supports automatic window layout of subwindows vertically on the screen but not horizontally across the screen.

4.4.1 Tool Creation

The client controls the number and type of subwindows available in the Tool's window. By using the **Tool.Make*SW** routines the client can have Tajo create a vanilla subwindow, differentiate it to be of type * and add it to the Tool. There are routines in the Tool interface that make it possible to include client defined subwindows in a Tool.

4.4.2 Tool States

The state of a Tool is a result of the following user illusion projected by Tajo. A Tool can be in one of three states: *normal*, in which the user has access to the full functionality and user interface provided by the Tool; *tiny*, in which the Tool's window is displayed as a small, labelled icon, but in which the algorithmic functions of the Tool are still available (but potentially not user invokable because of the display change); and *inactive*, in which the Tool's window does not appear on the display and it is not functional. The user can cause transitions between these states by interacting with Tajo. The Tool interface helps to shield the client from is the changes in the Tool's state caused by the user.

It is the responsibility of the client to support the above model and to consume only those resources necessary for a particular state. Thus, if a Tool is tiny, it should not hang on to resources needed only for updating of the display since it is not being displayed (Tajo manages displaying the icon). If a Tool is inactive, it should additionally free all resources tied up in internal state (i.e. free all streams, turn off all communications packages, deallocate all storage from the system heap, etc). Since many of the resources belonging to a Tool are allocated by Tajo on the Tool's behalf, the client must understand what those resources are so that Tajo and the client don't both deallocate the same resource.

The Tool interface provides a degree of automation for these transition tasks. For instance, when a Tool is made tiny the Tool interface will get each of the subwindows to free up resources that they use only to paint on the display. When a Tool is made inactive Tajo deallocates the subwindows and their associated data structures, destroys any menus associated with the Tool and closes streams. The client cannot be totally oblivious to the state transitions, else it might try to use a subwindow, or other Tajo object, that had been deallocated. The client is asked to clean up his own private data structures.

5.0 IMPLEMENTATION COMPONENTS

This section deals with the actual procedures and data structures that are to be used in building Tools that run in Tajo. There are, of course, considerably more definitions inside the interface files than appear in this document. However, as discussed in Section 1, this document is THE specification for writers of Tools. It is the supported external interface for Tool writers (i.e., the defs files are not!). For completeness, some interfaces are included below that are not specific to Tajo, but which are necessary for writing Tools. They are flagged in the list below by a preceeding *.

This section is organized with each major subsection representing a definitions module. The following is a table of contents for the Tajo Definitions modules:

ToolWindow – Tajo window creation and manipulation.
TTYSW – Teletype like operation.
UserInput – User input/notification.
***UserTerminal** – Display terminal primitives.
***Window** – Window contents manipulation.
***WindowFont** – Font information primitives.

5.1 Caret

Tajo provides a simple mechanism for clients to implement and manage a blinking caret, i.e., insertion point. Actual positioning and marking is the clients responsibility.

First some definitions.

```
Action: TYPE = {clear, mark, invert, stop, reset, ...};
MarkProcType: TYPE = PROCEDURE [data: POINTER, action: Action];
```

To become the manager of the caret call

```
Set: PROCEDURE [data: POINTER, marker: MarkProcType];
```

data is passed back to **marker** whenever it is called by Tajo. If a client does not want to actually mark the display when it is the manager of the caret it can use **NopMarkerProc** as its **marker.**

```
NopMarkerProc: MarkProcType;
```

```
ActOn: PROCEDURE [Action];
```

This procedure allows clients to act upon the current caret without regard to who is the current owner.

```
ResetOnMatch: PROCEDURE [data: POINTER];
```

This procedure allows clients to relinquish control of the blinking caret if they are currently the owner. Note that simply doing a Set with a **marker** that is the **NopMarkerProc** does not work because of race conditions in an arbitrary pre-emption environment.

```
UniqueAction: PROCEDURE RETURNS [Action];
```

This procedure allows clients to define private actions. This implies that implementors of caret marking procedures should ignore actions they do not implement or understand.

5.2 CmFile

Tajo provides a simple set of procedures for processing "User.cm" format files. This format is defined by the **CmFile** implementation as follows. A "cm" file is a sequence of sections. A section is a title line followed by zero or more name-value pairs. A section may not have embedded blank lines because a blank line is considered to terminate a section. The title line begins with a "[" and the section title is defined to terminate with the first succeeding "]". Each name-value pair is on a separate line; the name can be preceeded only by white space, and must be followed by a ": ". The value can be preceeded only by spaces. Tajo does not support the simultaneous processing of cm files by multiple processes. If the client wishes to do such processing, the client must create a layer above CmFile with appropriate monitoring.

If you simply want one entry from a section the following procedure will suffice.

```
Line: PROCEDURE [fileName, title, name: STRING] RETURNS [STRING];
```

If you are planning to process multiple entries in a file you should open the file or section, process the entries and then close file.

```
Open: PROCEDURE [fileName: STRING];

OpenSection: PROCEDURE [fileName, title: STRING] RETURNS [BOOLEAN];

Close: PROCEDURE [fileName: STRING];
```

Close must be called if either **Open** or **OpenSection** is called, even if they fail.

The following procedure will efficiently and sequentially process all the entries in a User.cm section.

```
NextItem: PROCEDURE [name, args: STRING];
```

The STRINGs returned by **NextItem** and **Line** are allocated from the system storage heap. The responsibility for deallocating them is the caller's.

All the procedures mentioned thus far have equivalent procedures that have the same names prepended with "UserCm" that deal with the file *User.cm* without requiring the client to pass in a **fileName**.

The following procedures are provided to aid clients in parsing lines. Note that **Lop** returns **Ascii.NUL** when the **String.SubString** is exhausted.

```
Lop: PROCEDURE [ss: String.SubString] RETURNS [c: CHARACTER];
```

Similar to SAIL's Lop. Returns the first character of ss and fixes up the length and offest.

```
GetNextToken: PROCEDURE [source: String.SubString, token: STRING] RETURNS [valid: BOOLEAN];

GetNextTokenAsABoolean: PROCEDURE [source: String.SubString] RETURNS [b: BOOLEAN];

GetNextTokenAsANumber: PROCEDURE [source: String.SubString] RETURNS [i: INTEGER];
```

ReadLineOrToken reads from the stream **sh** until **terminator** is found, unless either end-of-line or end-of-stream is encountered. The resulting line or token is returned via **buffer** and the resulting CHARACTER is the break character. If **buffer** is too short, **ReadLineOrToken** quits and

the resulting CHARACTER is the character being processed when the **buffer** overflows.

```
ReadLineOrToken: PROCEDURE [
  sh: Compatibility.SHandle, buffer: STRING, terminator: CHARACTER]
  RETURNS [CHARACTER];
```

TitleMatch returns TRUE if and only if the contents of **buffer** is in the right format to be the start of the section specified by **title.**

```
TitleMatch: PROCEDURE [buffer, title: STRING] RETURNS [BOOLEAN];
```

The following SIGNAL is defined for this interface. It can be RESUMEd only if the **code** is **multipleOpens.**

```
Error: SIGNAL [code: ErrorCode];
ErrorCode: TYPE = {multipleOpens, noneOpen, notOpen};
```

> **multipleOpens** can be raised on calls to open a file. **noneOpen** can be raised on calls that read data. **notOpen** can be raised on calls to close a file.

5.3 Compatibility

The type definitions in this interface provide some source level compatibility between corresponding Alto/Mesa and Pilot types. A Tool writer must be cognizant of this interface's existence because other Tajo interfaces use these types.

In the Alto world:

```
FHandle: TYPE = SegmentDefs.FileHandle;

SHandle: TYPE = StreamDefs.StreamHandle;
```

In the Pilot world:

```
FHandle: TYPE = File.Capability;

SHandle: TYPE = Stream.Handle;
```

5.4 Context

In performing various functions a Tool may wish to save and retrieve state from one notification to the next. This is an immediate consequence of the notification scheme, for a Tool cannot keep its state in the program counter after responding to an event without stealing the processor. Thus, it becomes necessary for a Tool to explicitly store its state. Since most notification calls to a Tool provide a window or subwindow handle, it is natural to associate these *contexts* with windows. As an alternative to the Tool having to build an associative memory to retrieve its context given a window handle, the context mechanism is provided. The actual context implementation mechanism is hidden from clients.

First some definitions.

```
Data: TYPE = POINTER TO UNSPECIFIED;

DestroyProcType: TYPE = PROCEDURE [Data, Window.Handle]

Type: TYPE = {all, first, last};
```

Type is an open enumeration.

Now the procedures that deal with contexts.

```
UniqueType: PROCEDURE RETURNS [Type];
```

If a client needs a unique **Type** not already in use by either Tajo or another client.

```
Create: PROCEDURE [type: Type, data: Data, proc: DestroyProcType, window: Window.Handle];
```

Creates a new context of the specified type. The context is associated with the indicated window. If the window already possesses a context of the specified type, the ERROR **Error[duplicateType]** will be raised. The **proc** is supplied so that when the window is destroyed all of the context data can be destroyed (deallocated) in a straight forward and orderly manner.

To destroy a context of specific **type** on a window **window** call **Destroy**. If the context exists on the window, this will first result in the **DestroyProcType** for that context being called and then the context itself will be deallocated. To destroy all of the contexts on **window** call **DestroyAll**. DestroyAll can be very dangerous because Tajo keeps its window specific data in contexts on the window. DestroyAll should not be used except in special circumstances. It is called by the routines that destroy window.

```
Destroy: PROCEDURE [type: Type, window: Window.Handle]

DestroyAll: PROCEDURE [window: Window.Handle];

Find: PROCEDURE [type: Type, window: Window.Handle] RETURNS [Data];
```

Retrieves the **data** field from the specified context for the window. NIL is returned if no such context exists on the window.

The client can change the data pointed to by the **data** field of a context at any time. Note that this can lead to race conditions if multiple processes are doing **Find**'s for the same context. It is the client's responsibility to MONITOR the data in such cases. To change the **data** field itself, call **Set**. Subsequent **Find**'s will return the new data.

```
Set: PROCEDURE [type: Type, data: Data, window: Window.Handle];
```

This is a no-op if no such context exists on the window.

```
SimpleDestroyProc: DestroyProcType;
```

This procedure merely calls the system heap node deallocator on the **data** field and is provided for clients when their context data is a simple heap node.

The only error condition detected by contexts is if you attempt to create a context of a specific type and one of that type already exists the following **ERROR** is raised.

```
Error: ERROR [code: ErrorCode];
ErrorCode: TYPE = {duplicateType};
```

5.5 Cursor

The **Cursor** interface provides a procedural interface to the hardware mechanism that implements the cursor on the screen. In order for these routines to be effective, all Tools must manipulate the cursor through them, else chaos reigns.

5.5.1 The Cursor Object

The cursor facilities define a **Object** which is a portable representation of the data which would be the hardware data if this cursor was the current hardware cursor:

```
Object: TYPE = RECORD [
    info: Info,
    array: UserTerminal.CursorArray ];
Handle: TYPE = POINTER TO Object;

Info: TYPE = RECORD [
    type: Type,
    hotX: [0..16),
    hotY: [0..16) ];

Type: TYPE = {
    activate, blank, bullseye, confirm, crossHairsCircle, ftp, ftpBoxes, hourGlass, lib, menu,
    mouseRed, mouseYellow, mouseBlue, mtp, pointDown, pointLeft, pointRight, pointUp,
    questionMark, retry, scrollDown, scrollLeft, scrollLeftRight, scrollRight, scrollUp,
    scrollUpDown, textPointer, typeKey, last};
```

We draw a distinction between user and system manufactured cursors. To keep things straight we allow clients to access system cursors only by their type.

5.5.2 Manipulating the Cursor

Cursor objects are normally created and managed by Tajo using the following routines.

```
Defined: TYPE = Type[activate..typeKey];

Set: PROCEDURE [Defined];
```

The above procedure allows you to set the displayed cursor to be one of the system-defined cursors.

```
Store: PROCEDURE [Handle];
Swap: PROCEDURE [old, new: Handle];
```

These procedures allows a client to store a cursor of his own design.

Fetch: PROCEDURE [Handle];

Copies the current cursor object into the cursor object pointed to by **Handle.**

FetchFromType: PROCEDURE [cursor: Handle, type: Defined];

Copies the cursor object that constitutes **type** into the cursor object pointed to by **Handle.**

GetInfo: PROCEDURE RETURNS [Info];

This procedure lets you find out about the current cursor.

UniqueType: PROCEDURE RETURNS [Type];

This procedure lets clients assign a unique type to their defined cursors.

The cursors in the subrange **Type[activate..typeKey]** are built-in (system supplied). Some special notes on what the built-in cursors look like (in general the **Type** names are sufficient description):

activate - used by the Librarian interface to indicate that a libject is being activated, it says LIB in the upper half, ACT in the lower.

ftp - used to indicate a file transfer in progress, it says FTP along the diagonal from the upper left to the lower right, with triangles in the lower left and upper right corners.

ftpBoxes - also used to indicate a file transfer in progress, it has black quadrants in the upper left and lower right, white quadrants elsewhere.

lib - used to indicate a Librarian transaction in progress, it says LIB along the diagonal from the upper left to the lower right, with triangles in the lower left and upper right corners.

mouseRed, mouseYellow, mouseBlue - a picture of a mouse, with the appropriate button highlighted.

textPointer - just like the one in Bravo.

The following procedures are useful for user feedback functions. **Invert** makes each white bit in the current cursor black, and vice versa. It returns TRUE if the new state of the cursor is positive. **MakePositive** restores the current cursor's polarity to be as if a **Set** or **Store** had just been done, and **MakeNegative** is equivalent to **MakePositive** followed by **Invert.**

Invert: PROCEDURE RETURNS [BOOLEAN];
MakeNegative: PROCEDURE;
MakePositive: PROCEDURE;

5.6 Event

Client programs sometimes need to be notified when certain system global events occur so that they can perform some operation. The Event mechanism provides that facility. The implementation of this mechanism parallels the ImageDefs **Cleanup** mechanism in the Mesa System with these main differences:

Additions have been made to the list of **Reasons.**

The event list is run with interrupts and timeouts turned on.

For events that mean the current environment is being exited, a client can veto, i.e., cancel, the execution of the event. The client might do this if a Tool can't properly handle the event or there is some operation that the user should have an opportunity to perform before the event actually takes place (e.g. saving an edited file). Vetoing is not supported in the debugger.

5.6.1 Items

The types and data structures involved with client Event procedures are:

```
Notifier: TYPE = PROCEDURE [why: Reason];
VetoProc: TYPE = PROCEDURE [why: EndReasons] RETURNS [BOOLEAN];
```

Returns TRUE when the client wants to cancel the event.

```
Item: TYPE = RECORD [
    link: ItemHandle ← NIL,
    eventMask: WORD,
    eventProc: Notifier,
    vetoMask: WORD ← NullMask,
    vetoProc: VetoProc ← NIL];

ItemHandle: TYPE = POINTER TO Item;

Reason: TYPE = {
    newFiles,          -- A file has just been retrieved to or deleted from the disk.
    flushSymbols,      -- Any symbols cached in the debugger may no longer be valid.
    newSession,        -- A new debugging session is starting.
    resumeSession,     -- Just swapped from the client to the debugger & not newSession.
    resumeDebuggee,    -- About to swap into the client world from the debugger.
    abortSession,      -- User has just Killed from the debugger.
    stopMesa,          -- Some client is about to call ImageDefs.StopMesa.
    abort,             -- User has keyed Shift-Swat.
    makeImage,         -- About to make an image file.
    makeCheck,         -- About to make a checkpoint file.
    startImage,        -- Have just started an image file.
    restartCheck,      -- Have just started a checkpoint file.
```

```
    continueCheck,      -- Continuing to run after just having made a checkpoint file.
    setDefaults         -- Some system global default has changed, e.g., Profile value. -- };

EndReasons: TYPE = Reason [resumeDebuggee..makeImage];

Masks: ARRAY Reason OF WORD = [...];

NullMask: WORD = 0B;
```

When the Event notification mechanism is invoked, each item is examined. If the bit corresponding to the reason is set in **item.eventMask**, **item.eventProc** is called. To avoid unnecessary code swapping clients should set the mask field so that their procedure is invoked only for those events about which they wish to be notified. This also applies to **item.vetoMask** and **item.vetoProc.** For example, if a client needed to be notified when an image file was being made or started, the mask would be set to:

```
Masks[makeImage] + Masks[startImage]
```

The following procedures are used to add or remove an **Item.** The storage for the **Item** is the responsibility of the client.

```
AddNotifier: PROCEDURE [item: ItemHandle];

DropNotifier: PROCEDURE [item: ItemHandle];
```

Warning: These procedures should not be invoked from inside an **Event.Notifier** procedure.

5.6.2 Notification

A client runs the list of **Items** by calling **Notify**, which can not be invoked from inside an **Event.Notifier** procedure. The list of **Items** is reversed by **Notify** as it executes unless the **why** is **newFiles**, **flushSymbols** or **setDefaults.** A caller of **Notify** in which **why** is in **EndReason** should be prepared to catch **Vetoed.**

```
Notify: PROCEDURE [why: Reason];

Vetoed: SIGNAL;
```

Warning: a client that doesn't call **Notify** when it instigates an event may cause other Tools to fail. However, do not call **Notify** with a **why** other than **newFiles** or **setDefaults** without talking to a Tajo implementor.

5.7 FileSW

The interface **FileSW** provides the definitions and procedures to create text subwindows whose backing storage is a disk file plus procedures that are specific to file type subwindows. All non-file subwindow specific manipulations are contained in the interface **TextSW.**

First some definitions.

```
Access: TYPE = {read, append, edit}; --TextSource.Access;
EnumerateProcType: TYPE =
 PROCEDURE [sw: Window.Handle, name: STRING, access: Access]
 RETURNS[done: BOOLEAN];
Options: TYPE = TextSW.Options;
Stream: TYPE = TextSource.Stream;

defaultOptions: Options = [
 access: read,
 menu: TRUE,
 split: TRUE,
 wrap: TRUE,
 scrollbar: TRUE,
 flushTop: FALSE,
 flushBottom: FALSE];
```

The following procedure creates a disk source and a text subwindow using that disk source. If **s** is NIL then a stream is automatically created on the file **name.** If **s** is not NIL then **name** must be the name of the file that **s** is the stream on. The text is positioned so that the character specified by **position** is displayed on the first line of **sw.**

```
Create: PROCEDURE [
 sw: Window.Handle, name: STRING,
 options: Options ← defaultOptions,
 s: Stream ← NIL, position: TextSource.Position ← 0];
```

Clients can destroy a file subwindow by calling

```
Destroy: PROCEDURE [sw: Window.Handle];
```

The following procedure will enumerate all of the current file subwindows.

```
Enumerate: PROCEDURE [proc: EnumerateProcType];
```

The file name and stream that are currently in a file subwindow are returned by

```
GetFile: PROCEDURE [sw: Window.Handle] RETURNS [name: STRING, s: Stream];
```

IsIt returns TRUE if and only if a window is a file subwindow.

```
IsIt: PROCEDURE [sw: Window.Handle] RETURNS [yes: BOOLEAN];
```

A new file is loaded into a file subwindow by calling

```
SetFile: PROCEDURE [
  sw: Window.Handle, name: STRING,
  s: Stream ← NIL, position: TextSource.Position ← 0];
```

Clients that contruct their own menus may include the following menu command routine. It does the standard load operation using the current selection as the file name argument.

```
LoadMCR: Menu.MCRType;
```

Tajo provides simple file editing facilities. Clients may determine if a file subwindow is currently editable by calling

```
IsEditable: PROCEDURE [sw: Window.Handle] RETURNS [yes: BOOLEAN];
```

A file subwindow is made editable by calling

```
MakeEditable: PROCEDURE [sw: Window.Handle] RETURNS [ok: BOOLEAN];
```

PutEditableFile stores the edited file on the new file **name.** If **name** = NIL then the old version of the file is saved as "currentName$" and the edited file is output to currentName.

```
PutEditableFile: PROCEDURE [sw: Window.Handle, name: STRING];
```

To reset the edited file to its original state call **ResetEditableFile.** The file subwindow is not editable after the call.

```
ResetEditableFile: PROCEDURE [sw: Window.Handle];
```

All of the file subwindow procedures can generate one or more of the following error conditions.

```
ErrorCode: TYPE = {notAFileSW, isAFileSW, notEditable, accessDenied};
Error: SIGNAL [code: ErrorCode];
```

5.8 Format

All the procedures in the Format interface take a procedure that can be called with a STRING, a piece of data to be formatted and where appropriate, a format specification.

```
DateFormat: TYPE = {dateOnly, noSeconds, dateTime, full};

NumberFormat: TYPE = RECORD [
 base: [2..36],
 zerofill, unsigned: BOOLEAN,
 columns: [0..255]];
```

The string produced using this record as a format specification for number formatting is **columns** wide. If **columns** is 0 only the needed number of columns are used. Extra columns are filled with zeros if **zerofill** is true, otherwise spaces are used. The number is treated as unsigned if **unsigned** is true.

```
LongSubStringDescriptor: TYPE = RECORD [
  base: LONG STRING,
  offset, length: CARDINAL];
LongSubString: TYPE = POINTER TO LongSubStringDescriptor;

StringProc: TYPE = PROCEDURE [s: STRING];
```

The data is formatted into a string which is passed in the call to the **StringProc**. The types of data that can be formatted are reflected in the following procedures:

```
Char: PROCEDURE [char: CHARACTER, proc: StringProc];

Date: PROCEDURE [pt: Time.Packed, format: DateFormat, proc: StringProc];
```

The date format used is the Mesa system's. A **full** date is formatted into something that looks like " 1-Jun-78 14:56:01 PDT".

```
Decimal: PROCEDURE [n: INTEGER, proc: StringProc];

LongDecimal: PROCEDURE [n: LONG INTEGER, proc: StringProc];

LongNumber: PROCEDURE [n: LONG UNSPECIFIED, format: NumberFormat, proc: StringProc];

LongOctal: PROCEDURE [n: LONG UNSPECIFIED, proc: StringProc];
```

Appends a B to the string if the value is greater than 7.

```
LongString: PROCEDURE [s: STRING, proc: StringProc];

LongSubStringItem: PROCEDURE [ss: LongSubString, proc: StringProc];
```

If **ss.base** is large **proc** is called multiple times with pieces of the substring.

```
Number: PROCEDURE [n: UNSPECIFIED, format: NumberFormat, proc: StringProc];

Octal: PROCEDURE [n: UNSPECIFIED, proc: StringProc];
```

Appends a B to the string if the value is greater than 7.

```
SubString: PROCEDURE [ss: String.SubString, proc: StringProc];
```

If **ss.base** is large **proc** is called multiple times with pieces of the substring.

5.9 FormSW

The client constructs a Form subwindow by specifying an array of *form item handles.* Each handle points to an item; each item is a variant record which contains a pointer to the specific data to be displayed and altered. The item contains information about how and, optionally, where it should be displayed. When appropriate, the item also contains *notification procedures* that are called by the Form subwindow to inform the client of events affecting the item.

The client's items are displayed in a subwindow, and are alterable by the user at any time unless explicitly prohibited by the client. The Form subwindow supplies procedures (via the PNR mechanism) to display, select or alter any of these items.

Clients of this interface should keep in mind that forms can't be arbitrarily large due to sizable storage requirements. The fixed overhead in heap usage per form item is 23 words (broken down as follows: 4 words for the item record, 1 word for the handle, 8 words for the item's TextSource plus 1 word for heap overhead, and 9 words for the item's TextDisplay Object). The variable overhead is due to the STRINGs associated with an item (the tag for example), line tables associated with multi-line items and the variant part of the item record.

5.9.1 Some conventions

It is important to distinguish between the user actions of choice and selection. The user is said to select an item (or part of an item) if that action changes the current selection, otherwise the user is said to make a choice of (or in) the item. Note that it is often not possible to distinguish between the two cases by simply looking at the display marking actions.

Some parts of the FormSW interface describe positions within an item. These positions are always zero origin, starting to the left of the first character of the tag (or main body of the item, if there is no tag).

5.9.2 The ItemObject

The Tool may specify any of the following generic types of item:

- Commands
- Labels
- Numbers
- Sets
- Strings

The **ItemObject** is the fundamental data structure of the Form subwindow. Unfortunately, the **ItemObject** is complex in order to provide sufficient flexibility to the Tool writer who wants fine control over displaying and altering the item. Most clients should not explicitly construct an **ItemObject**, but should instead use the procedures that allocate an **ItemObject** and take advantage of the defaulting mechanism; see the sample Tools in the appendices for examples. In FormSW procedure types the argument is called **item** if it is an **ItemHandle** and **items** if it is an **ItemDescriptor**.

```
ItemObject: TYPE = RECORD [
   tag: STRING,
   place: Window.Place,
   flags: ItemFlags,
   body: SELECT type: ItemType FROM
      boolean => [...],
      command => [...],
      enumerated => [...],
      longNumber => [...],
      number => [...],
      string => [...],
      tagOnly => [...],
      ENDCASE ];

ItemFlags: TYPE = RECORD [
   readOnly: BOOLEAN ← FALSE,
   invisible: BOOLEAN ← FALSE,
   drawBox: BOOLEAN ← FALSE,
   hasContext: BOOLEAN ← FALSE,
   clientOwnsItem: BOOLEAN ← FALSE];

ItemType: TYPE = {
   boolean, command, enumerated, longNumber, number, string, tagOnly};
ItemHandle: TYPE = POINTER TO ItemObject;
ItemDescriptor: TYPE = DESCRIPTOR FOR ARRAY OF ItemHandle;

BooleanHandle: TYPE = POINTER TO boolean ItemObject;
CommandHandle: TYPE = POINTER TO command ItemObject;
EnumeratedHandle: TYPE = POINTER TO enumerated ItemObject;
LabelHandle: TYPE = TagOnlyHandle;
LongNumberHandle: TYPE = POINTER TO longNumber ItemObject;
NumberHandle: TYPE = POINTER TO number ItemObject;
StringHandle: TYPE = POINTER TO string ItemObject;
TagOnlyHandle: TYPE = POINTER TO tagOnly ItemObject;

nullItems: ItemDescriptor = DESCRIPTOR[NIL, 0];
nullIndex: CARDINAL = LAST[CARDINAL];
```

tag is a client supplied string that is displayed immediately preceeding the data associated with the parameter (e.g. "tag: string:"). It may be NIL, in which case any trailer characters that are usually displayed after the tag will be suppressed (e.g. ": ").

place is the x,y position (subwindow relative) where the tag and data are to be displayed if the subwindow is of type fixed, otherwise place is ignored (see procedural interface). The array of item pointers is required to have the places in ascending (English reading) order. If the x position is negative, it is treated as a relative offset, where the magnitude of x specifies the number of bits to

leave between the end of the preceeding item and the start of the tag for this item. Note that the use of a negative x following a string item with **defaultBoxWidth** results in the ERROR **ItemError[illegalCoordinate, i]**, where **i** is the index of the offending item. Similarly, negative y positions are interpreted specially. They are line positions, i.e. they specify position as a multiple of the line height for the subwindow. The constants **line0** through **line9** can be used as y values to specify that the item should be on the zeroth through ninth lines in the subwindow. The procedure **LineN** takes a line number and returns the appropriate negative y.

```
LineN: PROCEDURE [n: CARDINAL] RETURNS [INTEGER];
```

In addition, there are several special constants. **sameLine** specifies that the y position for this item should be the same as the y position for the preceding item. If this is the first item, the ERROR **ItemError[illegalCoordinate, ----]** results. **nextLine** specifies that the y position for this item should be the next line after the y position of the preceding item.

Two special places are provided. **nextPlace** specifies that this item should be on the same line as the preceding one, and should start a little past where that one left off. This is subject to all of the caveats mentioned for negative x's above. **newLine** specifies that this item should start on the next line down from the preceding item, and works even if there is no preceding item.

```
nextPlace: Window.Place = [-10, sameLine];
newLine: Window.Place = [0, nextLine];
```

It is often desirable to have the items on different lines have the same horizontal positions. To simplify this task the **SetTagPlaces** procedure is provided. The **tabStops** are in raster points if **bitTabs** is TRUE, otherwise they are multiplied by the width of the digit 0. A positive x is used as a zero origin index into the **tabStops** array. If the **place** is **nextPlace** it means move to the next tab stop. Negative x's are left alone. This routine is a pre-processor that changes the items' places; it should be called before giving the items to the FormSW package.

```
SetTagPlaces: PROCEDURE [
  items: ItemDescriptor, tabStops: DESCRIPTOR FOR ARRAY OF CARDINAL, bitTabs: BOOLEAN];
```

The height of a line can be determined by calling **LineHeight**, which accounts for all fudge factors added to the fontHeight.

```
LineHeight: PROCEDURE RETURNS [CARDINAL];
```

flags is a RECORD of state bits for the item. The meaning of the flags is

If **readOnly** is TRUE, the user cannot modify this parameter. If any modification is attempted, the **readOnlyNotifyProc** for this subwindow is called.

If **invisible** is TRUE, the item is not displayed in the subwindow, and it is treated by Form subwindows exactly as it is was not present, except that it is occupying an index slot.

If **drawBox** is TRUE, the item is displayed enclosed within a box that is one bit thick.

If **hasContext** is TRUE, a client context is associated with the item. This context serves the same function as a client context associated with a subwindow.

If **clientOwnsItem** is TRUE, the Form subwindow will not try to deallocate the item if the subwindow is destroyed.

The following sections describe the feedback and actions that are associated with each generic parameter type.

5.9.2.1 Commands

For **command** parameter items the character "!" is appended to the **tag** as an indication to the user that this is a command item. User choice of this type of parameter item causes invocation of the supplied client **proc** in a manner analogous to menu command choice. FormSW supplies **NopNotifyProc** that does nothing when called.

```
ItemObject: TYPE = RECORD [...
  body: SELECT type: ItemType FROM
    command => [proc: ProcType],
  ...];

ProcType: TYPE = PROCEDURE [
  sw: Window.Handle ← NIL, item: ItemHandle ← NIL,
  index: CARDINAL ← nullIndex];

NotifyProcType: TYPE = ProcType;
NopNotifyProc: NotifyProcType;
```

5.9.2.2 Sets

5.9.2.2.1 Booleans

For **boolean** parameter items there is no special trailer appended to the **tag**. User choice of this type of parameter item causes this sequence of actions: the tag is inverted on the display; the sense of the BOOLEAN pointed to by **switch** is inverted; and then the supplied client **proc** is invoked. FormSW supplies **NopNotifyProc** that does nothing when called.

```
ItemObject: TYPE = RECORD [...
  body: SELECT type: ItemType FROM
    boolean => [
      switch: POINTER TO BOOLEAN,
      proc: NotifyProcType],
  ...];
```

switch is a POINTER TO BOOLEAN so that the client need not have access to the **ItemObject** in order to have access to the BOOLEAN. Note that this requires that the BOOLEAN occupy its own word in memory. This can be achieved by allocating the BOOLEAN in the client's global frame (but not in a RECORD in the global frame unless it is a MACHINE DEPENDENT RECORD and the BOOLEAN is specified to occupy a word) or by using the overlaid variant

```
WordBoolean: TYPE = RECORD [
  SELECT OVERLAID * FROM
    f1 => [b: BOOLEAN],
    f2 => [w: WORD],
    ENDCASE ];
```

Of these solutions, the overlaid variant tends to be the most clumsy and should be avoided.

5.9.2.2.2 Enumerateds

For **enumerated** parameter items the special trailer ": {" is appended to the **tag.** In addition, a "}" is appended at the end of the item's display representation. User modification of this type of parameter item causes this sequence of actions: the display is updated, in a manner that depends upon the **feedback**; the UNSPECIFIED pointed to by **value** is updated to match the display; and then the supplied client **proc** is invoked. FormSW supplies **NopEnumeratedNotifyProc** that does nothing when called.

```
ItemObject:  TYPE = RECORD [...
  body: SELECT type: ItemType FROM
    enumerated => [
      feedback: EnumeratedFeedback,
      copyChoices: BOOLEAN,
      value: POINTER TO UNSPECIFIED,
      proc: EnumeratedNotifyProcType,
      choices: EnumeratedDescriptor],
  ...];
EnumeratedFeedback: TYPE = {all, one};
EnumeratedNotifyProcType: TYPE = PROCEDURE [
  sw: Window.Handle ← NIL, item: ItemHandle ← NIL,
  index: CARDINAL ← nullIndex, oldValue: UNSPECIFIED ← nullEnumeratedValue];
NopEnumeratedNotifyProc: EnumeratedNotifyProcType;
nullEnumeratedValue: UNSPECIFIED = LAST[CARDINAL];
```

Examples of the two forms of feedback are:

> **all** - The item displays as "tag: {a, b, c}". Choosing any item within the curly brackets video reverses that item and sets the **value** in the associated record.
>
> **one** - The item displays as "tag: {a}". Only the currently chosen value is displayed.
>
> [Note: An enumerated can never have an unknown value (unless the client is not playing by the rules). It may have **nullEnumeratedValue**, in which case the display of the item has nothing between the braces.]

For both forms, the items available for choice are those STRINGs supplied by the client in the **choices.** When the **string** from one of the **choices** is chosen, the corresponding **value** from the **Enumerated** is stored into **ItemObject.value↑.** Depressing the menu mouse button displays the set of strings available for choice.

```
EnumeratedDescriptor: TYPE = DESCRIPTOR FOR ARRAY OF Enumerated;
Enumerated: TYPE = RECORD [
  string: STRING, value: UNSPECIFIED];
```

value is a POINTER TO UNSPECIFIED so that the client need not have access to the **ItemObject** in order to have access to the UNSPECIFIED. This introduces the same problems that occur with the **boolean ItemObject**'s **switch**, and the same solutions and caveats apply here. **value** points to an UNSPECIFIED so that its possible values can be from any type (usually an enumeration).

The **copyChoices** BOOLEAN is TRUE iff Form subwindow believes that the client's **choices** were copied into The Heap. See 5.9.3 for further details.

Occasionally a Tool wants to display a BOOLEAN choice without using the **boolean ItemObject**'s display conventions. The procedure **BooleanChoices** and an **enumerated ItemObject** can be used in this case.

```
BooleanChoices: PROCEDURE RETURNS [EnumeratedDescriptor];
```

5.9.2.3 Strings

For **string** parameter items the characters ": " are appended to **the tag** as an indication to the user that this is a string item. String items give the Tool writer explicit control over the alteration of the supplied string and over how it is to be displayed. The Tool supplied procedures are called whenever characters are to be added to the **string**.

```
ItemObject: TYPE = RECORD [...
  body: SELECT type: ItemType FROM
    string => [
      feedback: StringFeedback,
      inHeap: BOOLEAN,
      string: POINTER TO STRING,
      boxWidth: CARDINAL,
      filterProc: FilterProcType,
      menuProc: MenuProcType],
  ...];
```

inHeap - If this BOOLEAN is TRUE, the Tajo **StringEditProc** will dynamically allocate and deallocate the backing string from The Heap.

string - This is a POINTER TO STRING that contains the characters entered by the user. The level of indirection is provide so that the original string may be replaced.

feedback - The characters of **string** are displayed on the screen as text unless **feedback** is **password**, in which case a "*" is printed in place of each character of **string**.

```
StringFeedback: TYPE = {normal, password};
```

boxWidth - This is added to the **tag**'s width (including the supplied trailer) in order to determine the width of the box in which the STRING is displayed. If the special value **defaultBoxWidth** is used, then the box will extend to the right edge of the subwindow or to the next item, whichever is closer.

filterProc - The client's **filterProc** is called whenever characters are input for a selected string item. It is the responsibility of this procedure to actually edit the string.

```
FilterProcType: TYPE = PROCEDURE [
   sw: Window.Handle, item: ItemHandle, insert: CARDINAL, string: STRING];
StringEditProc: FilterProcType;
```

string, which may be NIL, contains the characters to edit into the existing string at position **insert**. The actual edit must be performed by calling **StringEditProc** in order to maintain the consistency of the selection and insert, and in order to allow FormSW to optimize the display updating.

menuProc - The client's **menuProc** is called whenever the user selects the string item with the menu button. This gives the client the opportunity to supply a list of strings to be displayed in a menu.

```
Hints: TYPE = DESCRIPTOR FOR ARRAY OF STRING;
FreeHintsProcType: TYPE = PROCEDURE [hints: Hints];
MenuProcType: TYPE = PROCEDURE [sw: Window.Handle, index: CARDINAL]
  RETURNS [hints: Hints, freeHintsProc: FreeHintsProcType, replace: BOOLEAN];
```

If **replace** is TRUE, then when the user chooses an appropriate menu item (i.e. hint) it will replace the item's **string**'s contents. If **replace** is FALSE, then when the user chooses the menu item it will be inserted into the item's **string** just as if the user had typed the menu string. If BASE[**hints**] = NIL, no prompt menu will be available to the user. This condition holds if the **menuProc** is the FormSW supplied one

```
VanillaMenuProc: MenuProcType;
```

freeHintsProc is called to free the **hints**, allowing the **hints** to be somewhere other than in the client's global frame. Two standard **hints** deallocators are FormSW supplied. **InHeapFreeHintsProc** assumes that the **hints** are from The Heap, while **NopFreeHintsProc** does nothing (appropriate if the **hints** are in the client's global frame).

```
InHeapFreeHintsProc: FreeHintsProcType;
NopFreeHintsProc: FreeHintsProcType;
```

5.9.2.4 Numbers

The **number** and **longNumber** item types are identical except in some small and obvious ways. Only the **number** item is discussed in detail; the differences found in **longNumber** are enumerated.

For **number** parameter items the special trailer "= " is appended to the **tag**. The user can select and edit a **number** item just like a **string** item, and the client can also exercise control over the alteration and display of the item, similar to a **string** item.

```
ItemObject: TYPE = RECORD [...
  body: SELECT type: ItemType FROM
    longNumber => [
      signed, notNegative: BOOLEAN,
      radix: Radix,
      boxWidth: CARDINAL [0..256),
      proc: LongNumberNotifyProcType,
      default: LONG UNSPECIFIED,
      value: POINTER TO LONG UNSPECIFIED,
      string: STRING],
    number => [
      signed, notNegative: BOOLEAN,
      radix: Radix,
      boxWidth: CARDINAL [0..128),
      proc: NumberNotifyProcType,
      default: UNSPECIFIED,
      value: POINTER TO UNSPECIFIED,
      string: STRING],
  ...];
```

signed - FormSW needs to know whether or not to treat the value as a signed number (i.e. INTEGER). It is treated as an INTEGER iff **signed** is TRUE.

notNegative - The user is permitted to enter negative values iff **notNegative** is FALSE.

radix - If the user does not provide a specific radix, 'D for decimal or 'O for octal, when he enters or modifies the item, then the radix is assumed to be 10 if **radix** is **decimal**, 8 if **radix** is **octal**.

```
Radix: TYPE = {decimal, octal};
```

boxWidth - Just as for a **string** item.

proc - The client's **proc** is called after each user edit to the item. If the client is not interested in such notification, it can use the "do nothing" **NopNumberNotifyProc**.

```
NumberNotifyProcType: TYPE = PROCEDURE [
  sw: Window.Handle ← NIL, item: ItemHandle ← NIL,
  index: CARDINAL ← nullIndex, oldValue: UNSPECIFIED ← LAST[INTEGER]];
NopNumberNotifyProc: NumberNotifyProcType;
```

default - It is possible that the user will not wish to enter any value for the item. In this case, the value is forced to **default**.

value - is a POINTER TO UNSPECIFIED so that the client need not have access to the **ItemObject** in order to have access to the UNSPECIFIED. FormSW assumes that the UNSPECIFIED occupies a full word, hence it should not be declared by the client to be a subrange of CARDINAL or INTEGER. **value** points to an UNSPECIFIED so that it can be either a CARDINAL or an INTEGER.

string - is the string representation of **value**↑. It is always be convertible to **value**↑ unless it is empty, in which case **value**↑ will be **default.**

The **longNumber** parameter item differs in that: **boxWidth** should be larger; **value** points to a LONG UNSPECIFIED instead of an UNSPECIFIED; **default** is a LONG UNSPECIFIED instead of an UNSPECIFIED; **proc** is of different type, and the FormSW supplied "do nothing" procedure is also different.

```
LongNumberNotifyProcType: TYPE = PROCEDURE [
  sw: Window.Handle ← NIL, item: ItemHandle ← NIL,
  index: CARDINAL ← nullIndex,
  oldValue: LONG UNSPECIFIED ← LAST[LONG INTEGER]];
NopLongNumberNotifyProc: LongNumberNotifyProcType;
```

5.9.2.5 Labels and Tags

The **tagOnly** item type is provided for two purposes. The first is to act as a label for some part of the form; for example, a form might consist of two parts, one for specifying input parameters and the other for output parameters. The client could distinguish the individual items by having their **tags** prefixed by "Input-" or "Output-", or it could have two sets of items with the same **tags** but preceded by a labelling line consisting of an item whose **tag** was "Input parameters" or "Output parameters". The second purpose is to substitute for the **tag** of a **string** item. This is useful when the client wishes to present the illusion that the **tag** for an item is not on the same line as the item's body. It is these two styles of usage that motivate the types **LabelHandle** and **TagOnlyHandle.**

```
ItemObject:  TYPE = RECORD [...
  body: SELECT type: ItemType FROM
    tagOnly => [
      sw: Window.Handle,
      otherItem: CARDINAL],
  ...];
```

sw - This is the FormSW that contains the item; it is automatically set by **Create**; clients should ignore it.

otherItem - This is the index of the other item which this item is acting as a tag for. If it is **nullIndex**, then the **tagOnly** is treated as a label instead of a substitute tag. If **otherItem** is not **nullIndex**, it must be the index of a **string** item or the ERROR **ItemError[notStringOtherItem, i]** will be generated by **Create**, where **i** is the index of this item.

In order to allow a **tagOnly** to act as a substitute tag, there is no special trailer appended to the tag. When a **tagOnly** item is used as a substitute tag, all of the user actions directed at its tag are redirected by FormSW to the **otherItem**. Due to this redirection the notification procedures of the target **string** item are called with arguments identical to the ones provided by FormSW when the **string** item's tag is operated on by the user.

5.9.3 ItemObject allocation and deallocation

5.9.3.1 Allocating an ItemObject from the Heap

The allocation procedures always allocate from the system Heap, using the standard facilites provided by Storage. There is no provision made for the client to provide an alternative allocator to FormSW. Allocation procedures always return a differentiated **ItemHandle**; ItemObjects allocated this way occupy a node only big enough for the specific variant allocated.

A call to an allocation procedure looks like a record constructor, with some of the fields found in an **ItemObject** omitted. Each field is defaulted if a reasonable default exists. An allocated item always has a FALSE **clientOwnsItem**.

```
BooleanItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace,
    proc: NotifyProcType ← NopNotifyProc,
    switch: POINTER TO BOOLEAN]
    RETURNS [BooleanHandle];

CommandItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace,
    proc: ProcType]
    RETURNS [CommandHandle];

EnumeratedItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace,
    feedback: EnumeratedFeedback ← one,
    proc: EnumeratedNotifyProcType ← NopEnumeratedNotifyProc,
    copyChoices: BOOLEAN ← TRUE,
    choices: EnumeratedDescriptor,
    value: POINTER TO UNSPECIFIED]
```

```
    RETURNS [EnumeratedHandle];

LabelItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace]
    RETURNS [LabelHandle];

LongNumberItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace,
    signed: BOOLEAN ← TRUE,
    notNegative: BOOLEAN ← FALSE,
    radix: Radix ← decimal,
    boxWidth: CARDINAL [0..256) ← 64,
    proc: LongNumberNotifyProcType ← NopLongNumberNotifyProc,
    default: LONG UNSPECIFIED ← LAST[LONG INTEGER],
    value: POINTER TO LONG UNSPECIFIED]
    RETURNS [LongNumberHandle];

NumberItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace,
    signed: BOOLEAN ← TRUE,
    notNegative: BOOLEAN ← FALSE,
    radix: Radix ← decimal,
    boxWidth: CARDINAL [0..128) ← 64,
    proc: NumberNotifyProcType ← NopNumberNotifyProc,
    default: UNSPECIFIED ← LAST[INTEGER],
    value: POINTER TO UNSPECIFIED]
    RETURNS [NumberHandle];

StringItem: PROCEDURE [
    tag: STRING ← NIL,
    readOnly, invisible, drawBox, hasContext, inHeap: BOOLEAN ← FALSE,
    place: Window.Place ← nextPlace,
    feedback: StringFeedback ← normal,
    boxWidth: CARDINAL ← defaultBoxWidth,
    filterProc: FilterProcType ← StringEditProc,
    menuProc: MenuProcType ← VanillaMenuProc,
    string: POINTER TO STRING]
    RETURNS [StringHandle];
```

```
TagOnlyItem: PROCEDURE [
   tag: STRING ← NIL,
   readOnly, invisible, drawBox, hasContext: BOOLEAN ← FALSE,
   place: Window.Place ← nextPlace,
   otherItem: CARDINAL ← nullIndex]
   RETURNS [TagOnlyHandle];
```

5.9.3.2 Deallocating an ItemObject from the Heap

An item allocated by FormSW can be deallocated by calling

```
FreeItem: PROCEDURE [item: ItemHandle] RETURNS [ItemHandle];
```

If **clientOwnsItem** is TRUE, then the actions taken (which differ for each item type) are:

enumerated - If **copyChoices** is TRUE, the **choices** are freed.

longNumber, number - The **string** is freed.

string - If **inHeap** is TRUE, the **ItemObject.string** is freed.

All other types - Nothing is freed.

If **clientOwnsItem** is FALSE, then the actions taken are those for when it is TRUE, but in addition the **tag** and the item are also freed. The **ItemHandle** returned by **FreeItem** is NIL if **clientOwnsItem** is FALSE, otherwise it is the argument.

5.9.4 Subwindow Global Operations

The Tool writer creates a Form subwindow by calling

```
Create: PROCEDURE [
 sw: Window.Handle,
 clientItemsProc: ClientItemsProcType,
 readOnlyNotifyProc: ReadOnlyProcType ← IgnoreReadOnlyProc,
 options: Options ← [],
 initialState: ToolWindow.State ← active];

ClientItemsProcType: TYPE = PROCEDURE
 RETURNS [items: ItemDescriptor, freeDesc: BOOLEAN];
ReadOnlyProcType: TYPE = ProcType;

Type: TYPE = {fixed, relative};
Options: TYPE = RECORD [type: Type ← fixed, scrollVertical: BOOLEAN ← TRUE];
```

sw - the subwindow that is transformed into a Form subwindow. If the subwindow is already a Form subwindow, the ERROR **Error[alreadyAFormSW]** results.

clientItemsProc - this will be called at FormSW's discretion to get the **items.** If the **ItemDescriptor** was manufactured from The Heap, perhaps by calling **AllocateItemDescriptor**, then the client can have FormSW free it by returning a TRUE **freeDesc.**

```
AllocateItemDescriptor: PROCEDURE [CARDINAL] RETURNS [ItemDescriptor];
```

readOnlyNotifyProc - the client procedure that is called whenever the user attempts to modify an item with a TRUE **readOnly** flag. Two standard **ReadOnlyProcTypes** are supplied by FormSW: **IgnoreReadOnlyProc** blinks the display when called, while **NopReadOnlyProc** simply does nothing.

```
ReadOnlyProcType: TYPE = ProcType;
IgnoreReadOnlyProc: ReadOnlyProcType;
NopReadOnlyProc: ReadOnlyProcType;
```

options - a **type** of **relative** directs the Form subwindow to automatically determine where and how the items and their associated data are displayed. If the client specifies **fixed** then the client must designate a subwindow place for each item to be displayed; it is the client's responsibility to avoid overlapping or overwriting of items and their data. If **scrollVertical** is TRUE, a vertical scrollbar is provided.

[Note: In the relative case the parameter items are simply displayed one per line. This implies that the height of a subwindow that would contain all of your parameters is = n*LineHeight[].]

initialState - This determines whether the Form subwindow is awake when created. If **initialState** is not **active**, then the Form subwindow will be asleep. If **initialState** is **active**, then the **clientItemsProc** is called while still in **Create.**

You may transform a Form subwindow back into an undifferentiated subwindow by calling

```
Destroy: PROCEDURE [Window.Handle];
```

If the subwindow is not currently a Form subwindow, the ERROR **Error[notAFormSW]** results. It is possible to test whether a subwindow is in fact a Form subwindow by calling

```
IsIt: PROCEDURE [sw: Window.Handle] RETURNS [yes: BOOLEAN];
```

If it is necessary to move the subwindow within the parent window, or to change its size, the adjustment is done by calling

```
Adjust: ToolWindow.AdjustProcType;
```

The current **Options** are changed by calling

```
SetOptions: PROCEDURE [sw: Window.Handle, options: Options];
```

Display is provided to allow the Tool to re-display the contents of the subwindow. Note that **Display** allows the Tool to scroll, or unscroll, the items before the redisplay via the **yOffset**, which specifies the number of bits to offset the items upwards

```
Display: PROCEDURE [sw: Window.Handle, yOffset: CARDINAL ← 0];
```

If the Tool window is being made tiny, there is no need for its subwindows to keep state information that is used only for display purposes. A Form subwindow can be told to discard such state data by calling **Sleep**, and to recreate the display state (when the window becomes big) by calling **Wakeup.**

```
Sleep: PROCEDURE [Window.Handle];
Wakeup: PROCEDURE [Window.Handle];
```

A Tool often wishes to know how high a Form subwindow should be to just display all of the items, assuming that they are not scrolled. There are two heights of interest; the minimum height for the subwindow is attained if none of the textual item types (i.e. **longNumber, number, string, source**) overflow a single line; the current height is the true height of the subwindow, accounting for overflowing items. These are returned by the following procedure as **min** and **current** respectively.

```
NeededHeight: PROCEDURE [Window.Handle]
  RETURNS[min, current: CARDINAL];
```

Unfortunately, **NeededHeight** requires that the Form subwindow already exist. It is occasionally convenient to know the height a Form subwindow would need at a minimum without having it exist yet. This number can be ascertained by calling

```
MinHeight: PROCEDURE [items: ItemDescriptor, type: Type] RETURNS [CARDINAL];
```

5.9.5 Operations Affecting One or Two Items

The following procedure is provided to allow the Tool to re-display the contents of an individual item. Redisplaying a single item may cause other items to also be redisplayed.

```
DisplayItem: PROCEDURE [sw: Window.Handle, index: CARDINAL];
```

The following procedures allow a Tool to get and set the currently selected item and the item containing the insert point. These procedures should be used judiciously so as to not preempt the user.

```
GetSelection: PROCEDURE [Window.Handle]
  RETURNS [index: CARDINAL, first, last: CARDINAL];
GetTypeIn: PROCEDURE [Window.Handle]
  RETURNS [index: CARDINAL, position: CARDINAL];
SetCurrent: PROCEDURE [sw: Window.Handle, index: CARDINAL];
SetSelection: PROCEDURE [
  sw: Window.Handle, index: CARDINAL, first, last: CARDINAL];
SetTypeIn: PROCEDURE [
  sw: Window.Handle, index: CARDINAL, position: CARDINAL];
```

nullIndex is used as an index when the client wants "nothing" selected or wants no insert point. **SetCurrent** is equivalent to **SetSelection** with **first** and **last** selecting the non-tag and trailer portion of the item; it also places the insert point at the item's end.

FormSW assumes that there is a unique mapping between an item and an index into the **ItemDescriptor** for each subwindow. The client notification procedures are called with the item, or it's index, or both. If only one is provided, FormSW provides a way to get the other. Given an item, it is possible to find its index by calling

```
FindIndex: PROCEDURE [sw: Window.Handle, item: ItemHandle] RETURNS [CARDINAL];
```

To go in the other direction, call

```
FindItem: PROCEDURE [sw: Window.Handle, index: CARDINAL] RETURNS [ItemHandle];
```

5.9.6 Errors and Abnormal Conditions

The following are all the ERRORs generated directly by FormSW.

```
Error: SIGNAL [code: ErrorCode];
ErrorCode: TYPE = {alreadyAFormSW, notAFormSW};

ItemError: SIGNAL [code: ItemErrorCode, index: CARDINAL];
ItemErrorCode: TYPE = {illegalCoordinate, notStringOtherItem, nilBackingStore};
```

The **index** argument to **ItemError** is the index of the item that FormSW was processing when it discovered the error condition.

5.10 HeapString

HeapString provides operations for STRINGs that are allocated from the system Heap. This assumption allows the procedures to provide automatic allocation of initially NIL string values and automatic expansion or shortening when required.

AppendChar: PROCEDURE [p: POINTER TO STRING, c: CHARACTER];

Appends the character **c** to the string pointed to by **p.**

AppendExtensionIfNeeded: PROCEDURE [to: POINTER TO STRING, extension: STRING]
RETURNS [BOOLEAN];

Checks the passed string pointed to by **to** to see if it contains an extension (contains a period followed by at least one character. If not; it appends **extension** (but does not supply a period!)

AppendString: PROCEDURE [to: POINTER TO STRING, from: STRING, extra: CARDINAL ← 0];

Appends the string **from** to the string pointed to by **to.** If the string must be expanded, it will be expanded to the new required length plus **extra.**

Replace: PROCEDURE [to: POINTER TO STRING, from: STRING];

Replaces the string pointed to by **to** with the string **from.**

5.11 Keys

Keys defines the user input devices' key layouts. It depends heavily on the KeyStations interface which defines the bits generated by the microcode for each key station. There are five types of key stations: 1) typing keys such as alphanumerics, punctuation, tab, CR, etc.; 2) function keys such as the left, right, and top function groups; 3) the mouse buttons; 4) the keyset paddles; and 5) diagnostic pseudo-keys for hardware diagnostic purposes.

```
DownUp: TYPE = {down, up};
KeyBits: TYPE = PACKED ARRAY KeyName OF DownUp;
```

Each element of the **KeyName** enumeration is a **KeyStations.KeyStation.** Refer to the definitions files if you need to know the exact bit assigned to a particular key station.

```
KeyStation: TYPE = [0..112);

KeyName: TYPE = MACHINE DEPENDENT {
    Keyset1, Keyset2, Keyset3, Keyset4, Keyset5,
    Red, Blue, Yellow,
    Five, Four, Six, E, Seven, D, U, V, Zero, K, Dash, P, Slash, BackSlash, LF, BS, Three, Two,
        W, Q, S, A, Nine, I, X, O, L, Comma, Quote, RightBracket, Spare2, Spare1, One,
        ESC, TAB, F, Ctrl, C, J, B, Z, LeftShift, Period, SemiColon, Return, Arrow, DEL, FL3,
        R, T, G, Y, H, Eight, N, M, Lock, Space, LeftBracket, Equal, RightShift, Spare3, FL4,
        FR5, R5, R9, L10, L7, L4, L1L, A9, R10, A8, L8, L5, L2, R2, R7, R4, D2, D1, Key48,
        T1, T3, T4, T5, T6, T7, T8, T10, R3, Key47, A10, R8};
```

There are some common synonyms defined for use on the Alto II keyboard.

```
FL1: KeyName = DEL;
FL2: KeyName = LF;
BW: KeyName = Spare1;
FR1: KeyName = Spare3;
Swat: KeyName = FR1;
FR2: KeyName = BackSlash;
FR3: KeyName = Arrow;
FR4: KeyName = Spare2;
```

5.12 Librarian

This set of procedures is the lowest level of interface to the Librarian Server.

Procedures in this interface fall into three basic categories as follows.

Altering The Librarian Data Base
Interrogating The Librarian Data Base
Accessing the *Contents* of Libjects

First, some types:

```
Card13: TYPE = [0..17777B];
LibjectID: TYPE = LONG CARDINAL;
LibjectVersionType: TYPE = {version, replacementID, timeAndDate};
LibjectVersion: TYPE = MACHINE DEPENDENT RECORD [
    type: LibjectVersionType,
    body: SELECT LibjectVersionType COMPUTED FROM
        version => [
            compatablity: CARDINAL,
            addition: CARDINAL,
            modification: CARDINAL,
            patch: CARDINAL];
        replacementID => [
            pad: Card13,
            id: LibjectID,
            zip: CARDINAL ← 0];
        timeAndDate => [
            pad: Card13 ← 17776B,
            tod: LONG CARDINAL,
            zip: CARDINAL ← 0];

LibjectUpdateType: TYPE = {compatability, addition, modification, patch};
FullLibjectIDHandle: TYPE = POINTER TO FullLibjectID;
FullLibjectID: TYPE = RECORD [id: LibjectID, version: LibjectVersion];

SnapShot: TYPE = ARRAY [0..1) OF FullLibjectID
SnapShotHandle: TYPE = DESCRIPTOR FOR ARRAY OF FullLibjectID
```

5.12.1 Altering The Librarian Data Base

```
LibjectCreate: PROCEDURE [s: STRING] RETURNS [id: LibjectID];
```

Ensures that the supplied string is not in use or is not a hash collision in the **LibjectID** space, then marks it as used and returns that **LibjectID.** Fine point: THE name of a Libject is its LibjectID! The string is associated with the Libject for annotation and other purposes.

Callers of LibjectCreate should be prepared to handle the error codes **AlreadyExists** and **IDConflict.**

The librarian data base, for any instance of a Librarian service, can grow arbitrarily. The actual data base is maintained on a network file service, where the number and size of files is not a problem. The Librarian service maintains, on local disk storage, a cache of the most recently accessed Libjects. Libjects that are in the cache are said to be *active.* The following procedures allow clients to explicitly activate and deactivate Libjects.

```
LibjectActivate: PROCEDURE [id: LibjectID, wait: BOOLEAN ← FALSE];

LibjectDeactivate: PROCEDURE [id: LibjectID];
```

Procedures that access the Librarian data base take an **activate** BOOLEAN. The Librarian interface automatically activates a Libject if **activate** is TRUE and the error code **InactiveLibject** is encountered during the execution of the procedure.

```
LibjectCheckout: PROCEDURE [
    fid: FullLibjectIDHandle, reason: STRING, activate: BOOLEAN ← FALSE];
```

Marks the Libject as checked-out. This is essentially a write lock with the added feature of supplying a STRING which can be examined by other clients of the database.

```
LibjectCheckin: PROCEDURE [
    id: FullLibjectIDHandle, updatetype: LibjectUpdateType, pl: PropertyList,
    s: Compatiblity.SHandle, activate: BOOLEAN ← FALSE];
```

Creates a new version of the Libject from the supplied property list (described below). Clears the checked-out lock. If the stream is non-NIL, then the contents of the stream will be stored at the destination specified by the ContentsFile property (with a new version number), and the ContentsFile property updated.

The client need not know the algorithm for updating libject versions. However, the client is asked to specify his intended updatetype via the following enumerated set and the interface will compute the next version number.

[Note: The stream is destroyed by the procedure.]

5.12.2 Interrogating the Librarian Data Base

```
LibjectFindID: PUBLIC PROCEDURE [s: STRING, activate, wait: BOOLEAN ← FALSE]
  RETURNS [id: LibjectID];
```

Performs the string to **LibjectID** conversion.

```
LibjectFindVersion: PROCEDURE [id: LibjectID, s: SnapShotHandle, activate: BOOLEAN ← FALSE]
  RETURNS [fid: FullLibjectIDHandle];
```

Will supply the right version of the Libject for the supplied snapshot.

The code used to create a **snapShotHandle** that means *current version* follows:

```
BEGIN OPEN Libraian;
snap: SnapShot;
snapshot: SnapShotHandle ← DESCRIPTOR[snap];
snap ← [[id: AllFromHereID, version: [version, version[tod: Time.Current[]]]]];
END;
```

```
LibjectHeaderLook: PROCEDURE [
  fid: FullLibjectIDHandle, pl: PropertyList, activate: BOOLEAN ← FALSE];
```

This is the basic question/answer procedure. It fills-in the property-values in the supplied property list.

5.12.3 Accessing the *Contents* of Libjects

The term *contents* is used to mean the file that the libject is keeping track of (e.g. mesa source files, text files, etc.).

```
LibjectContentFile: PROCEDURE [
    id: FullLibjectIDHandle, localname: STRING, activate: BOOLEAN ← FALSE]
    RETURNS [Compatibility.FHandle];
```

Gives the caller a handle to the file that contains the contents of the specific Libject-version. It makes a copy (via some file transfer facility) of the contents of the specified Libject on the local volume and gives it the supplied **localname.**

```
LibjectContentStream: PROCEDURE [id: FullLibjectIDHandle, activate: BOOLEAN ← FALSE]
  RETURNS [Compatibility.SHandle];
```

Gives the caller a handle to a stream on the contents of the specified Libject-version. It initially makes a copy (via some file transfer facility) of the contents of the specified Libject on the local volume. It is not clear whether ultimately it should copy the file or page it over the network. In either case, Tajo's local storage vanishes when the stream is destroyed.

5.12.4 Errors and Abnormal Conditions

In general, most error conditions are non-resumable and recovery should be handled by the caller outside of a catch phrase. The responsibility of the **Librarian** interface is to clean up on UNWINDs.

```
Error: SIGNAL [code: ErrorCode, message: STRING, pl: PropertyList];
```

Unusual conditions encountered by the Librarian Interface are reported to the client via the SIGNAL **Error.** This SIGNAL has three arguments: **code**, an enumerated type that is to be interpreted by the client; an optional (may be NIL) STRING, **message**, to be interpreted by the client; and an optional (may be empty) **PropertyList, pl**, that supplies more detailed information about the abnormal condition (normally only of interest to the Librarian Interface).

[Fine Point: The storage for the parameters **message** and **pl** is *owned* by the Librarian Interface (i.e. the Librarian Interface is responsible for its allocation and destruction). This means that if a client wishes to make use of the values of these parameters outside the catching phrase he must make local copies.]

```
ErrorCode: TYPE = {CheckedOut, NotCheckedOut, WrongVersion, ServerDead,
    InvalidServerName, AlreadyExists, UnassignedID, IDConflict, FTPError, BufferOverFlow,
    InactiveLibject, UnknownError};
```

5.12.5 Property Lists

The basic mechanism used for data communication (and its type information) with the Librarian Interface (and ultimately the Librarian Data Base) is via a record called a ***PropertyList.*** Understanding the syntax and semantics of these records and the operations upon them is essential to conversing with the Librarian Interface. The Librarian Interface makes no effort to hide anything from clients of **PropertyLists.**

```
PropertyNumber: TYPE = RECORD [
  prefix: [0..3777B],
  type: PropertyValueType];
```

with the types of the values being the enumerated type

```
PropertyValueType: TYPE = {
  TwoWord, LibjectID, LibjectIDARRAY, String, Record, PropertyList};
```

PropertyPairs are defined as a variant record as follows.

[note that all **VARIANTs** are three (3) words long! This is not accidental and may cause troubles as new **PropertyValueTypes** are invented or needed].

```
PropertyPairHandle: TYPE = POINTER TO PropertyPair;
PropertyPair: TYPE = RECORD [
    empty: BOOLEAN,
    pn: PropertyNumber,
    body: SELECT COMPUTED PropertyValueType FROM
        TwoWord => [value: Inline.LongNumber],
        LibjectID => [id: LibjectID],
        LibjectIDArray, Record => [
            length: CARDINAL,
            ptr: POINTER],
        String => [
            length: CARDINAL,-- length of storage block (words)
            string: STRING],
```

```
    PropertyList => [pl: PropertyList],
    -- add new type definitions as needed
    ENDCASE];
```

The BOOLEAN field **empty** is useful for constructing lists of property pairs whose **PropertyNumber's** are correct but whose values are incorrect or *empty*. This feature is used, for example, to construct lists to be *filled-in*.

Finally a **PropertyList** is simply a descriptor for an array of **PropertyPairs**. [Caution: each element in an array of variant records will be the length of the *largest* variant.]

```
PropertyList: TYPE = DESCRIPTOR FOR ARRAY OF PropertyPair;
```

5.12.6 Property List Operations

The following procedures are supplied for manipulating PropertyLists.

```
CreatePropertyList: PROCEDURE [n: CARDINAL] RETURNS [PropertyList];
```

Allocates the storage necessary for a **PropertyList** of LENGTH[**n***SIZE[**PropertyPair**]] and initializes the **PropertyPairs** to have **empty** be TRUE.

```
DestroyPropertyList: PROCEDURE [plist: PropertyList]
```

Destroys a **PropertyList** (i.e. both the **PropertyList** contents and actual list storage is released). [The implementation of these two procedures uses the Mesa Heap allocation package.]

```
ResetPropertyList: PROCEDURE [plist: PropertyList]
```

Calls **ResetPropertyPair** for all non-empty values in the supplied **PropertyList.**

```
ValidatePropertyList: PROCEDURE [plist: PropertyList] RETURNS [size: CARDINAL]
```

Simple checking of the **PropertyList** is performed and its size in words is computed and returned if the **PropertyList** is valid, otherwise the SIGNAL **InvalidPropertyList** is raised.

5.12.7 PropertyPair Operations

```
ResetPropertyPair: PROCEDURE[pair: PropertyPairHandle]
```

The contents of a **PropertyPair** are released. [The structure of a PropertyList tree is preserved across Reset. This means resetting a PropertyList PropertyPair does not release the PropertyList "pointed to"]

```
AddPropertyPair: PROCEDURE [plist: PropertyList, pp: PropertyPair] RETURNS [CARDINAL]
```

Allows you to set the values in a **PropertyList.** It searches the **PropertyList** for the first empty **PropertyPair** and sets it to the passed **PropertyPair.**

Users of the above procedure must be prepared to handle the SIGNAL **PropertyListFull.**

```
FindPropertyPair: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
  RETURNS [pp: PropertyPairHandle]
```

Finds the specified **PropertyNumber** in the **PropertyList** and returns its **PropertyPairHandle.** If the **PropertyNumber** cannot be found, a NIL is returned.

```
GetPropertyPair: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
  RETURNS [pp: PropertyPairHandle]
```

Like **FindPropertyPair** but will signal **PropertyNotFound** instead.

The following procedures construct **PropertyPair**s. These procedures allocate storage and copy the contents where necessary.

```
MakeEmptyPair: PROCEDURE [pn: PropertyNumber] RETURNS [PropertyPair];
MakeStringPair: PROCEDURE [pn: PropertyNumber, s: STRING] RETURNS [PropertyPair];
MakeRecordPair: PROCEDURE [pn: PropertyNumber, length: CARDINAL, ptr: POINTER]
 RETURNS [PropertyPair];
```

The following procedures are controlled LOOPHOLEs.

```
GetPropertyID: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
 RETURNS [pp: POINTER TO LibjectID PropertyPair, id: LibjectID];
GetPropertyList: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
 RETURNS [pp: POINTER TO PropertyList PropertyPair, pl: PropertyList];
GetPropertyRecord: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
 RETURNS [pp: POINTER TO Record PropertyPair, p: POINTER];
GetPropertyString: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
 RETURNS [pp: POINTER TO String PropertyPair, STRING, s: STRING];
GetPropertyTwoWord: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
 RETURNS [pp: POINTER TO TwoWord PropertyPair, ln: Inline.LongNumber];
GetPropertyValue: PROCEDURE [plist: PropertyList, pn: PropertyNumber]
 RETURNS [pp: POINTER TO TwoWord PropertyPair, lowbits, highbits: UNSPECIFIED];
```

The following procedures will *pack(unpack)* a **PropertyList** into(from) a contiguous block of storage of size 2+**ValidatePropertyList** words. They are useful for storing **PropertyList**s on files.

```
BundleOfBitsFromPropertyList: PROCEDURE [plist: PropertyList, p: POINTER];
```

Packs the specified **PropertyList** into a contiguous block of storage. It is the client's responsibility to supply a block large enough for the passed **PropertyList.**

```
PropertyListFromBundleOfBits: PROCEDURE [p: POINTER]
 RETURNS [PropertyList];
```

. Unpacks a packed **PropertyList**. Verifies that the passed pointer really points to a packed **PropertyList** and if not generates an ERROR.

5.13 Menu

One of the primary command execution invocation mechanisms in Tajo is the menu. The **Menu** Interface gives the Tool writer control over what menus the user will see and what actions an individual menu item will perform. How menus appear to the user and how he interacts with them is built-in and not of concern to the client of this interface.

5.13.1 Simple Creation of Menus

The following two procedures are designed to allow clients to make and free menus and menu items simply. For those who need to know more about menus, subsequent sections explain menus and their implementation in detail.

```
Make: PROCEDURE [
    name: STRING,
    strings: DESCRIPTOR FOR ARRAY OF STRING,
    mcrProc: MCRType,
    copyStrings: BOOLEAN ← TRUE,
    permanent: BOOLEAN ← FALSE]
    RETURNS [Handle];
```

Creates menu named **name** that has the elements contained in **strings**. When one of the strings is selected the **mcrProc** will be called indicating the index of the string in the array. The **permanent** flag indicates whether the created object can subsequently be destroyed. The **copyStrings** flag indicates whether **strings** should be copied into the system Heap.

```
Free: PROCEDURE [menu: Handle, freeStrings: BOOLEAN ← TRUE];
```

Frees the menu, optionally freeing the copied strings.

Menus are chosen for display based upon which window the cursor is over. This allows the displayed menu stack to be dynamic relative to the window layout. The following two procedures allow clients to associate menus with windows.

```
Instantiate: PROCEDURE [menu: Handle, window: Window.Handle];
```

Associates the menu with the passed window and increments a use count in **menu**. If this is the first menu to be instantiated in **window** the system global menu(s) is also instantiated. If **menu** is NIL only the system global menu is instantiated. If **menu** is already instantiated the ERROR **Error[alreadyInstantiated]** is generated.

```
Uninstantiate: PROCEDURE [menu: Handle, window: Window.Handle];
```

This procedure removes **menu** from the window and decrements its use count. Eventual deallocation of the menu is left to the Tool to perform. If this menu is not instantiated with this window, then the ERROR **Error[notInstantiated]** is generated. It is also possible that the ERROR **Error[contextNotAvailable]** will be generated; this indicates that Tajo has detected an internal inconsistency in its data structures.

5.13.2 The Menu Object

The **Object** contains the normally invariant data associated with a menu.

```
Handle: TYPE = POINTER TO Object;
Object: TYPE = RECORD [
    ~ ~ ~ ~ ~
    name: STRING
    items: Items
    ~ ~ ~ ~ ~ ];
Items: TYPE = DESCRIPTOR FOR ARRAY OF ItemObject;
```

5.13.3 Menu Instances

An unlimited number of menus may be associated (instantiated) with the tool window or any subwindow. The menu mechanism maintains a ring of menu instances (pointers to associated menus) for each subwindow (if there is at least one associated menu). One of these associated menus is taken to be the "current" menu for that subwindow.

Some menus (at least the system global ones) want to be available from virtually every subwindow. This could be accomplished by creating an **Object** for each use, but the primary memory cost of multiple copies of an **Object** is large. Additionally, some users may want to dynamically alter the items contained in menus (e.g. lists of available fonts, etc.). These requirements lead to the use of a level of indirection. Thus Tajo never copies a client's **Object**, instead it always keeps a pointer back to that **Object**. It is the client's responsibility to guarantee that the **Object** is valid as long as Tajo has a pointer to it.

5.13.4 Menu Items

Each menu item has a **keyword** (a string of characters). A menu item has a *Menu Command Routine (MCR)* associated with it. A MCR is a procedure that is called when the user specifies its corresponding item. Clients have found that using one MCR per menu is useful because only one large catch phrase need be written to handle common exception cases.

```
ItemObject: TYPE = RECORD [keyword: STRING, mcrProc: MCRType];

ItemHandle: TYPE = POINTER TO ItemObject;

MCRType: TYPE = PROCEDURE [
  window: Window.Handle ← NIL, menu: Handle ← NIL, index: CARDINAL ← LAST[CARDINAL]];
```

The following two procedures allow clients to make and free menu items;

```
MakeItem: PROCEDURE [keyword: STRING, mcrProc: MCRType] RETURNS [ItemObject];
FreeItem: PROCEDURE [ItemObject];
```

5.13.5 Procedures For Setting up Menus

The following procedures allow the Tool to create and destroy menus, to specify which menu selection techniques (in addition to the standard one) will be used, and to specify what procedures will be invoked (called) when a menu item is selected. The procedures that actually do the *work* of implementing how the human user sees and uses menus is not the concern of the Tool.

The following two procedures allow the Tool to create and destroy menus.

```
Create: PROCEDURE [items: Items, name: STRING, permanent: BOOLEAN] RETURNS [Handle];
```

> Returns a pointer to a menu **Object** named **name** that is made up of **items.** The **permanent** flag indicates whether the created object can subsequently be destroyed.

```
Destroy: PROCEDURE [Handle];
```

> Deallocates storage for the **Object** pointed to by **Handle.** It first verifies that the **Object** has a use count = 0, if not the ERROR **Error[isInstantiated]** is generated. If the menu is permanent, the ERROR **Error[isPermanent]** is generated.

The above procedures set up the data stuctures required for menu operations. They do not, however, set up a specific window's PNR for invoking menus. If the window is one managed by Tajo the standard menu PNR is already set up. If not, the client may set the standard menu PNR by calling

```
SetPNR[window: Window.Handle];
```

If it is necessary to set the menu PNR under a different mouse button than the one used by **SetPNR**, the PNR itself is accessible as **PNR.**

The following two procedures allow the Tool to get a handle for and to set the font used for menus.

```
GetFont: PROCEDURE RETURNS [font: WindowFont.Handle];
SetFont: PROCEDURE [font: WindowFont.Handle];
```

5.13.6 Errors and Abnormal conditions

```
Error: ERROR [code: ErrorCode];
ErrorCode: TYPE = {
 isInstantiated, alreadyInstantiated, notInstantiated, contextNotAvailable, isPermanent};
```

5.14 MsgSW

The **MsgSW** interface provides a simple way of posting messages to the user. A MsgSW is built upon a StringSW.

5.14.1 Creation/Destruction

To create a MsgSW call:

```
Create: PROCEDURE [
 sw: Window.Handle,
 lines: CARDINAL ← 1,
 options: TextSW.Options ← defaultOptions];
defaultOptions: TextSW.Options = [access: append, menu: TRUE, split: TRUE,
 wrap: TRUE, scrollbar: TRUE, flushTop: FALSE, flushBottom: FALSE];
```

The **lines** parameter specifies the minimum number of lines that the subwindow will keep in its backing store before discarding the oldest line. The subwindow height controls how many lines will be visible to the user. If the number of lines visible to the user is greater than **lines**, then all the visible lines are kept in the backing store.

When the **options.access** parameter is anything but **append** an **Error** is raised with a code of **appendOnly**.

The following procedure destroys the backing store and MsgSWness of the subwindow.

```
Destroy: PROCEDURE [sw: Window.Handle];
```

5.14.2 Output

A message line is delimited by a carriage return on its end. The latest message has a severity associated with it.

```
Severity: TYPE = {info, warning, fatal};
```

Here are the procedures for managing the contents of a MsgSW.

```
Post: PROCEDURE [
    sw: Window.Handle,
    string: STRING,
    severity: Severity ← info,
    prefix: BOOLEAN ← TRUE,
    endOfMsg: BOOLEAN ← TRUE];
```

Appends **string** onto the latest message. The severity of the message is **severity**. If the **prefix** parameter is **TRUE** and the message is starting a new line, a short string that depends on **severity** (**info**: "", **warning**: "Warning: " or **fatal**: "Fatal Error: ") starts the line before the client message. The **endOfMsg** parameter set to **TRUE** delimits the message without having to put an **Ascii.CR** in **string**.

```
PostAndLog: PROCEDURE [
    sw: Window.Handle,
    string: STRING,
    severity: Severity ← info,
    prefix: BOOLEAN ← TRUE,
    endOfMsg: BOOLEAN ← TRUE,
    logSW: Window.Handle ← NIL];
```

Like **Post** but with the additional **logSW** parameter that enables that same message that appears in the MsgSW to be directed to another subwindow for logging. If the value is NIL then the output is directed to the **UserInput.GetDefaultWindow[]** and the Tool's name is prepended to the message.

```
AppendString: PROCEDURE [window: Window.Handle, string: STRING];
```

Appends **string** onto the latest message. This is the procedure used for **UserInput.StringOut**. The severity is set to **info**.

```
Clear: PROCEDURE [sw: Window.Handle];
```

Erases the contents of the MsgSW. The severity is set to **info**.

The various message posting routines impart a severity to the current message. The following procedure sets it explicitly:

```
SetSeverity: PROCEDURE [sw: Window.Handle, severity: Severity];
```

5.14.3 Status Retrieval

These procedures provided status information about the current message:

```
GetSeverity: PROCEDURE [sw: Window.Handle] RETURNS [severity: Severity];
```

```
LastLine: PROCEDURE [sw: Window.Handle, ss: String.SubString];
```

The parameter **ss** is filled in with base, offset and length of the current message. The client may want to copy **ss** and the string **ss.base** since this information is liable to change.

5.15 Profile

The **Profile** interface provides an interface to a number of commonly accessed user and system data items. Note that all these items are read only.

bitmap: READONLY Window.Box;

The current size and position of the bitmap.

debugging: READONLY BOOLEAN;

TRUE if debugging. Used internally by Tajo to decide whether to attempt error recovery or call the debugger. If Tajo invokes the debugger, it may not be possible to continue the session.

librarian: READONLY STRING;

The current name of the Librarian server being used in librarian transactions.

registry: READONLY STRING;

The currently logged in user's mail registry.

userName: READONLY STRING;
userPassword: READONLY STRING;

The currently logged in user's name and password strings.

Tajo provides a Tool called the ProfileTool that allows users to set or alter these values. The **TajoMisc** interface provides procedures for clients to modify these items.

If a client needs to notice when one of these values has changed, either by the user or another client, it should have an **Event.Notify** procedure that detects the **setDefaults** event.

5.16 Put

All the procedures in the **Put** interface take a **Window.Handle**, a piece of data to be formatted and, where appropriate, a format specification. The data is formatted by the Format mechanism and then output by a call to the **UserInput.StringOut** procedure associated with the **Window.Handle**. If the **Window.Handle** passed into any Put procedure is NIL the output is directed to the **UserInput.StringOut** procedure of **UserInput.GetDefaultWindow[]**. See the documentation on the **Format** interface for comments about actual output format of the following procedures:

```
Blanks: PROCEDURE [h: Window.Handle, n: CARDINAL ← 1];

Char: PROCEDURE [h: Window.Handle, char: CHARACTER];

CR: PROCEDURE [h: Window.Handle];

CurrentSelection: PROCEDURE [h: Window.Handle];

Date: PROCEDURE [h: Window.Handle, pt: Time.Packed, format: Format.DateFormat];

Decimal: PROCEDURE [h: Window.Handle, n: INTEGER];

Line: PROCEDURE [h: Window.Handle, s: STRING];

LongDecimal: PROCEDURE [h: Window.Handle, n: LONG INTEGER];

LongNumber: PROCEDURE [
  h: Window.Handle, n: LONG UNSPECIFIED, format: Format.NumberFormat];

LongOctal: PROCEDURE [h: Window.Handle, n: LONG UNSPECIFIED];

LongString: PROCEDURE [h: Window.Handle, s: LONG STRING];

LongSubString: PROCEDURE [h: Window.Handle, ss: Format.LongSubString];

Number: PROCEDURE [h: Window.Handle, n: UNSPECIFIED, format: Format.NumberFormat];

Octal: PROCEDURE [h: Window.Handle, n: UNSPECIFIED];

SubString: PROCEDURE [h: Window.Handle, ss: String.SubString];

Text: PROCEDURE [h: Window.Handle, s: STRING];
```

5.17 Scrollbar

The scrollbar interface does not do scrolling, i.e., moving of bits on the screen, but rather provides a consistent user interface and mechanism for specifying and invoking scroll actions.

First some definitions.

```
Type: TYPE = {horizontal, vertical};
Direction: TYPE = {forward, backward, relative};
Percent: TYPE = [0..100];
```

Two types of procedures are used to perform the scroll operation. **ScrollProcType** procedures are used to communicate to the client a user's scroll request. **ScrollbarProcType** procedures are used to get the scrollbar data from the client in order to display them to the user.

```
ScrollProcType: TYPE = PROCEDURE
    [window: Window.Handle, direction: Direction, percent: Percent];
ScrollbarProcType: TYPE = PROCEDURE [Window.Handle]
    RETURNS[box: Window.Box, offset, portion: Percent];
```

Scrollbars may be created for vertical or horizontal scroll functions by the following procedure.

```
Create: PROCEDURE
    [window: Window.Handle,
    type: Type,
    scroll: ScrollProcType,
    scrollbar: ScrollbarProcType];
```

If **Create** is called for a subwindow that already has scrollbars of that **type** the following ERROR is generated.

```
Error: ERROR [code: ErrorCode];
ErrorCode: TYPE = {alreadyExists};
```

Scrollbars are deleted by calling

```
Destroy: PROCEDURE [sw: Window.Handle, type: Type];
```

The following procedure should be called when scollbars are to be invoked, normally when the cursor exits a window. The **UserInput.DefaultCursorPNR** already does this.

```
InvokeScrollbar: PROCEDURE [window: Window.Handle, place: Window.Place];
```

5.18 Selection

The **Selection** interface is the mechanism used to communicate the *current selection* among the various Tools. It is the responsibility of clients of this interface to provide for actual selection of text and/or graphics within its window(s). The client window that contains the current selection is refered to as the manager of the current selection.

Basically, any client wishing to become the manager of the current selection supplies the selection interface with a pair of procedures.

```
ActOnProcType: TYPE = PROCEDURE [data: POINTER, action: Action];
ConvertProcType: TYPE = PROCEDURE [data: POINTER, target: Target] RETURNS [POINTER];
```

The **ActOnProcType** is called if someone wishes to modify the current selection. The **ConvertProcType** is called if someone wants the *value* of the current selection.

The following procedure sets these procedures for the current selection.

```
Set: PROCEDURE [pointer: POINTER, ConvertProcType, actOn: ActOnProcType];
```

These requests are communicated to the manager of the current selection by passing an **Action** or **Target** which are defined as follows.

```
Action: TYPE = {clear, delete, mark, unmark, replace};
Target: TYPE = {window, subwindow, string, source, length, position};
```

The above definitions are presented as exact enumerations when in fact the are defined as open enumerations. The following two procedures allow clients to define their own private conversion types.

```
UniqueAction: PROCEDURE RETURNS [Action];
UniqueTarget: PROCEDURE RETURNS [Target];
```

Clients may act on the current selection, independent of who is the current owner, by calling

```
ActOn: PROCEDURE [Action];
```

The following useful specializations are also supplied.

```
Clear: PROCEDURE = INLINE BEGIN ActOn[clear]; END;
Delete: PROCEDURE = INLINE BEGIN ActOn[delete]; END;
```

It is some times difficult to determine if your are the manager of the current selection. The following procedure will clear the current selection iff you are the current owner. You are the current owner if **pointer** is equal to the latest **pointer** that was passed into the **Set**.

```
ClearOnMatch: PROCEDURE [pointer: POINTER];
```

The following procedure will perform the requested conversion and return a **POINTER** to the data. The data returned for items larger that one word is is allocated out of the system Heap with the storage ownership passed to the recipient which must deallocate it. **Target**s of **window** and **subwindow** return a one word **Window.Handle**. A **Target** of **string** returns a STRING allocated from the Heap. **Target**s of **length** and **position** return POINTER TO LONG CARDINAL.

```
Convert: PROCEDURE [Target] RETURNS [POINTER];
```

The manager of the current selection may chose not to implement some (or all) of the possible conversions. In that case, it simply returns NIL.

The following special conversions are supplied as a convenience to clients. If the current selection is not acceptable to the Mesa runtime as a number, then **String.InvalidNumber** will be raised by the runtime and allowed to propogate through these procedures.

```
Number: PROCEDURE [radix: CARDINAL ← 10] RETURNS [CARDINAL];

LongNumber: PROCEDURE [radix: CARDINAL ← 10] RETURNS [LONG CARDINAL];
```

5.18.1 Selection Sources

The selection interface defines the **Source** mechanism for processing textual selections that are more than a few hundred characters in length.

```
Source: TYPE = POINTER TO SourceObject;
SourceObject: TYPE = RECORD [
    data: POINTER TO UNSPECIFIED,
    proc: SourceProc,
    destroy: DestroyProc];
DestroyProc: TYPE = PROCEDURE [source: Source];
SourceProc: TYPE = PROCEDURE [data: POINTER, string: STRING];
```

The source mechanism works as follows:

The client asks for the current selection to be converted as a **source.**

The owner of the current selection creates an instance of the **Source** data structure and returns it to the client.

The client then makes repeated calls on **proc** supplying a string of arbitrary size.

The owner of the current selection fills the string with text and returns. The owner need not fill the string completely but it must return some data with each call as end-of-selection is indicated by returning an empty string.

When the client receives a zero length string it must call the **destroy** procedure supplied in the source object, otherwise the space allocated for the source will be lost.

5.18.2 Trash Bin

The selection interface defines an abstraction, somewhat similar to the current selection, know as the *trash bin.* The trash bin is used to save the most recent text cuts for subsequent pastes. Clients may become owners of the trash bin by calling

```
ClearTrashBinProcType: TYPE = PROCEDURE [data: POINTER];

SetTrashBin: PROCEDURE [
    pointer: POINTER, ConvertProcType, clear: ClearTrashBinProcType];
```

Clients can convert the contents of the trash bin in the same manner as the current selection by calling

```
ConvertTrashBin: PROCEDURE [Target] RETURNS [POINTER];
```

5.19 StringSW

The interface **StringSW** provides the definitions and procedures to create and manipulate text subwindows whose backing store is a **STRING**. Further text subwindow operations are described in the **TextSW** interface.

First some definitions.

```
Options: TYPE = TextSW.Options;
defaultOptions: Options = [access: edit, menu: TRUE, split: TRUE, wrap: TRUE,
 scrollbar: TRUE, flushTop: FALSE, flushBottom: FALSE];
```

A StringSW is create by calling

```
Create: PROCEDURE [
 sw: Window.Handle, s: POINTER TO STRING ← NIL
 options: Options ← defaultOptions];
```

> If **s** is **NIL** or **s↑** is **NIL**, the subwindow will allocate and manage a heap string for the backing store; otherwise, the client is responsible for the storage management of the string.

A subwindow's StringSW properties are destroyed by calling

```
Destroy: PROCEDURE [sw: Window.Handle];
```

The current backing **STRING** for a string subwindow is returned by

```
GetString: PROCEDURE [sw: Window.Handle] RETURNS [s: POINTER TO STRING];
```

Clients can determine if a window is a string subwindow by calling

```
IsIt: PROCEDURE [sw: Window.Handle] RETURNS [yes: BOOLEAN];
```

5.20 TajoMisc

The interface **TajoMisc** is a catch-all for public and semi-public Tajo utilities that did not fit logically into any of the other interfaces.

Part of Tajo's runtime state can be deduced from the following variable:

initialToolStateDefault: ToolWindow.State;

> This is the state in which a Tool is created if it does not override the default provided in the **Tool.Create** call.

To change the values in the Profile interface the following six procedures are used. Each of these procedures call **Event.Notify[why: setDefaults]**. STRINGs passed in as arguments are copied.

SetBitmap: PROCEDURE [box: Window.Box];

> Actually changes the bitmap to be **box**, relative to the screen. Check Profile.bitmap for the actual box used.

SetUserName: PROCEDURE [s: STRING ← NIL];
SetUserPassword: PROCEDURE [s: STRING ← NIL];

> In the AltoTajo, these procedures change **OsStaticDefs.OsStatics**. Also, if **s** = NIL then the **Profile** string is copied from **OsStaticDefs.OsStatics**.

SetDebugging: PROCEDURE [b: BOOLEAN];
SetLibrarian: PROCEDURE [s: STRING];
SetRegistry: PROCEDURE [s: STRING];

The following procedure must be used rather than **UserTerminal.SetState** to change the state of the display bitmap because **UserTerminal.SetState** bypasses Tajo with disasterous consequences.

SetState: PROCEDURE [new: UserTerminal.State] RETURNS [old: UserTerminal.State];

The next two procedures allow a process to do a WAIT for a period of time without having to be in a convenient MONITOR.

WaitMilliSecs: PROCEDURE[msec: CARDINAL];

WaitSecs: PROCEDURE[sec: CARDINAL];

If a client wishes to stop Tajo and all other Tools in a safe manner it should call **Quit**. This lets the normal cleanup mechanisms run which are responsible for making files on the disk consistent, etc.

Quit: PROCEDURE;

To load a Tool **name** from the disk:

CreateTool: PROCEDURE [name: STRING];

5.21 TextSource

The interface **TextSource** provides the primary underlying mechanism for the representation of data that is used to implement Tajo's uniform text display, selection and editing facilities. **TextSource** defines the standard set of operations that are sufficient for all access to a text source. Specific implementations may use additional operations for setting or altering the state of a text source. Text sources transmit information in atomic units called blocks.

```
Position: TYPE = LONG CARDINAL;
nullPosition: Position = LAST[LONG CARDINAL];
```

5.21.1 Basic Operations

The procedures which create text sources return a **Handle** which is an object oriented pointer to a record of procedures which defines the operations on a text source.

```
Handle: TYPE = POINTER TO Procedures;

Procedures: TYPE = POINTER TO ProceduresObject;

ProceduresObject: TYPE = RECORD [
 readText: ReadTextProc,
 deleteText: DeleteTextProc,
 insertText: InsertTextProc,
 getLength: GetLengthProc,
 setLength: SetLengthProc,
 actOn: ActOnProc,
 readBlock: ReadBlockProc];
```

Following are the definitions and semantics of each operation on a text source.

```
ActOnProc: TYPE = PROCEDURE [source: Handle, action: Action] RETURNS [ActionResult];
```

Actions defined for text sources are as follows:

```
Action: TYPE = {destroy, mark, sleep, truncate, wakeup};
```

destroy - free all storage and release all resources associated with the text source instance.

mark - mark the text source in some distinctive manner at its current end.

sleep - the text source will not be used for a while (hint to reduce current resources for this text source).

truncate - truncate the text source at it's current end. This has a noticable effect only for sources that have some representation in a file system.

wakeup - the text source is going to be used (undo what you did for sleep).

[Note: sleep and wakeup are only hints for storage and resource management purposes. This means that implementors must be able to handle all operations on sleeping text sources.]

All **ActionProcs** return an **ActionResult** to indicate the status of the requested action.

```
ActionResult: TYPE = {ok, new, bad};
```

Text is deleted in a text source by invoking **deleteText.**

```
DeleteTextProc: TYPE = PROCEDURE [
 source: Handle, position: Position, n: LONG CARDINAL, trash: BOOLEAN]
 RETURNS [realPos: Position, realN: LONG CARDINAL];
```

The **deleteText** operation is a request to delete the specified range of characters. The text source implementor may refuse to delete any (read or append) or only an initial subset of the requested interval. In any event the returned **realN** is the actual number of characters deleted (if any) and **realPos** is the index into the text source from which one should update the display representation.

The current length of the text source is obtained by the **getLength** operation. This operation is used extensively and its implementation should be efficient.

```
GetLengthProc: TYPE = PROCEDURE [source: Handle] RETURNS [Position];
```

Text is inserted into a text source by invoking the **insertText** operation.

```
InsertTextProc: TYPE = PROCEDURE [
 source: Handle, ss: String.SubString, position: Position] RETURNS [Position];
```

The returned **Position** is the index into the text source from which one should update the display representation.

Text is read from a text source by invoking either the **readText** or **readBlock** operation.

```
Block: TYPE = RECORD [
 base: POINTER TO PACKED ARRAY [0..0) OF CHARACTER,
 offset, length: CARDINAL];
Class: TYPE = {none, cr, alpha, space, other};

ReadBlockProc: TYPE = PROCEDURE [
 source: Handle, position: Position, maxLength: CARDINAL, class: Class]
 RETURNS [b: Block, pos: Position];

ReadTextProc: TYPE = PROCEDURE [
 source: Handle, position: Position, maxLength: CARDINAL, class: Class]
 RETURNS [ss: String.SubString, pos: Position];
```

ss.base is not a real STRING so don't reference the **length** or **maxlength** fields in it.

The basic semantics of these two procedures is: return a sequence of text that is either **maxLength** long or is terminated by a character of the specified **class.**

The following additional semantic rules for reading text sources are designed to ease the job of implementing a text source and to facilitate the implementation of *discontinuous* sources. Discontinuous sources are text sources that either have holes in them or contain sequences of non-textual data embedded in them (e.g. Bravo files that contain formatting).

A text source may not return more text than was requested.

A single call on read may not return text that is not contiguous in the text source's address space (i.e. it cannot concatinate two discontiguous runs of text).

A text source may return less text than was requested.

A text source may only return no text (i.e. length = 0) if the position is equal to the value returned by **getLength** or **pos** is greater than **position.**

Clients can determine if a character is in a given **class** by calling
```
TestClass: PROCEDURE [char: CHARACTER, class: Class] RETURNS [equal: BOOLEAN];
```

Clients can set the length of a text source by calling the **setLength** operation. For most sources attempting to lengthen a source by this operation is undefined and will produce unexpected results.

```
SetLengthProc: TYPE = PROCEDURE [source: Handle, position: Position]
 RETURNS [Position];
```

5.21.2 Useful Operations on TextSources

To append a portion of a source to a string call
```
Append: PROCEDURE [string: STRING, source: Handle, start: Position, n: CARDINAL];
```

The following collection of procedures work for all text sources.

The following two procedures are useful for identifying words and lines in text sources.

```
ScanType: TYPE = {word, line};
ScanLeft: PROCEDURE [source: Handle, start: Position, type: ScanType]
 RETURNS [left: Position];
ScanRight: PROCEDURE [source: Handle, start: Position, type: ScanType]
 RETURNS [right: Position];
```

The following procedure will scan a subrange of a text source looking for a string match.

```
TextSearch: PROCEDURE [
 source: Handle, string: STRING,
 start: Position ← 0, stop: Position ← LAST[LONG CARDINAL]]
 RETURNS [lineStart, left: Position];
```

In the event that **string** is not found the following error is raised.

```
SearchFailed: ERROR;
```

The following procedures are useful for processing substrings and doing simple edits such as backchar and backword while typing.

```
CharIsEditChar: PROCEDURE [char: CHARACTER] RETURNS [BOOLEAN];
DoEdit: PROCEDURE [source: Handle, editChar: CHARACTER, editPos: Position]
 RETURNS [delta: INTEGER];
```

The return argument **delta** is the count of the number of characters altered by the edit (e.g. the number of characters deleted by backword).

The following two procedures are used to identify edit and control characters in **ss**. On the return, **ss** is divided into **ss** (the part of the **ss** before the returned character) and **theRest** (the part of the original **ss** following the returned character).

```
FindEditChar: PROCEDURE [ss, theRest: String.SubString]
 RETURNS [editChar: CHARACTER, found: BOOLEAN];

FindControlChar: PROCEDURE [ss, theRest: String.SubString]
 RETURNS [char: CHARACTER, found: BOOLEAN];
```

5.21.3 Disk Sources

Disk file type text sources can be created with read or append access only. This is because mid file modifications (direct editing) are not supported. Disk file sources are created, with the specified access, by calling

```
Access: TYPE = {read, append, edit};

Stream: TYPE = Compatibility.SHandle;

CreateDisk: PROCEDURE [name: STRING, access: Access, s: Stream ← NIL]
 RETURNS [source: Handle];
```

If **s** is not NIL then it is used instead of getting a stream from the file named **name**.

Clients may determine the disk file name and the backing stream for a disk type text source by calling.

```
DiskInfo: PROCEDURE [source:Handle] RETURNS [name: STRING, s: Stream, access: Access];
```

A currently existing disk source can be renamed by calling the following procedure. The current disk source is destroyed and a disk source for the new file, with the specified **access**, is created.

```
RenameDisk: PROCEDURE [
 source: Handle, newName: STRING, access: Access, newSN: BOOLEAN]
 RETURNS [Handle];
```

5.21.4 String Sources

String type text sources can be created with read, append or edit access. String sources are created, with the specified access, by calling.

```
CreateString: PROCEDURE [ps: POINTER TO STRING, expandable: BOOLEAN]
  RETURNS [source: Handle];
```

[Note: The current implementation of string sources requires a contiguous block of memory large enough to entirely contain the backing string. More importantly, when the string is expanded a new larger string will be allocated and then the string copied, requiring 2*n+delta characters of memory. Beware of LARGE string sources]

Clients may determine the backing string currently in use and its state by calling

```
StringInfo: PROCEDURE [source:Handle]
  RETURNS [ps: POINTER TO STRING, expandable: BOOLEAN];
```

The following two procedures will edit strings.

```
DeleteSubString: PROCEDURE [ss: SubString, keepTrash: BOOLEAN]
  RETURNS [trash: STRING];
```

keepTrash indicates whether the deleted text is to be put into the global trash bin. (See Selection interface)

```
InsertString: PROCEDURE [
  string: POINTER TO STRING, position: CARDINAL, toAdd: SubString, extra: CARDINAL];
```

A string can be designated non-expandable by using the following special value for **extra.**

```
cannotExpand: CARDINAL = LAST[CARDINAL];
```

5.21.5 Errors

```
ErrorCode: TYPE = {fileNameError, accessError, isBad};
Error: SIGNAL [code: ErrorCode];
```

fileNameError - indicates either doesn't exist or bad syntax.

accessError - attempt to perform an operation that violates the created access option.

isBad - indicates that the source is no longer extant. This occurs on core swaps when the file is deleted.

5.22 TextSW

The TextSubwindow Package implements a comprehensive set of facilities for viewing text independent of source. This package essentially takes a client created subwindow and text source, creates the necessary data structures and then sets appropriate PNR's for viewing, scrolling, and text selection.

The available options for text subwindows are specified via the following options record.

```
Options: TYPE = RECORD [
 access: Access,
 menu: BOOLEAN,
 split: BOOLEAN,
 wrap: BOOLEAN,
 scrollbar: BOOLEAN,
 flushTop: BOOLEAN,
 flushBottom: BOOLEAN];

Access: TYPE = TextSource.Access;
```

If **menu** is TRUE, the standard text operations menu is instantiated with the subwindow at create time. If **split** is TRUE, the subwindow may be divided into an arbitrary number of *splits* or horizontal subregions by the user. If **wrap** is TRUE, a line that is too long to fit across the subwindow will be broken at a word boundary and continued on the next line, instead of terminating at the subwindow boundary. If **scrollbar** is true, the subwindow will be provided with a vertical scrollbar. The booleans **flushTop** and **flushBottom** specifiy if a standard border is to be supplied at the top and bottom of the subwindow.

If the client does not supply the options argument at create time the following defaults will be supplied.

```
defaultOptions: Options = [access: read, menu: TRUE, split: TRUE, wrap: TRUE,
 scrollbar: TRUE, flushTop: FALSE, flushBottom: FALSE];
```

5.22.1 Basic Operations

Text subwindows are created by calling

```
Create: PROCEDURE [
 sw: Window.Handle, source: TextSource.Handle,
 options: Options ← defaultOptions, position: Position ← 0];
```

> **position** indicates the initial character position in **source** that should be displayed at the top of the subwindow.

The following procedure destroyes a text subwindow, freeing all data structures and setting all PNRs to system defaults. However, the client supplied source is not destroyed.

```
Destroy: PROCEDURE [sw: Window.Handle];
```

> Attempting to destroy a non-text subwindow is a nop.

The following procedures allow the client to alter the contents of the text source currently being displayed in the text subwindow. The text subwindow and source must have either edit or append

access to correctly use these operations.

```
DeleteText: PROCEDURE [
  sw: Window.Handle, pos: Position, count: LONG CARDINAL];
```

Clients may insert text, at the current insertion position, by calling one of the following procedures.

```
InsertChar: PROCEDURE [sw: Window.Handle, char: CHARACTER];
InsertString: PROCEDURE [sw: Window.Handle, s: STRING];
InsertSubString: PROCEDURE [sw: Window.Handle, ss: String.SubString];
```

5.22.2 Positioning and Selection Operations

The following procedures allow clients to obtain and alter the current state of a text subwindows insertion, selection and "end-of-file" positions.

```
GetEOF: PROCEDURE [sw: Window.Handle] RETURNS [Position];
GetInsertion: PROCEDURE [sw: Window.Handle] RETURNS [Position];
GetSelection: PROCEDURE [sw: Window.Handle] RETURNS [left, right: Position];
SetEOF: PROCEDURE [sw: Window.Handle, eof: Position];
SetInsertion: PROCEDURE [sw: Window.Handle, position: Position];
SetSelection: PROCEDURE [sw: Window.Handle, left, right: Position];
```

The client can determine if any portion of the source is currently being displayed with the following procedure.

```
PositionIsVisible: PROCEDURE [sw: Window.Handle, position: Position] RETURNS [BOOLEAN];
```

The following procedure enables clients to *resolve* window coordinates to the nearest text source positions. It always returns a valid position.

```
PositionFromPlace: PROCEDURE [sw: Window.Handle, place: Window.Place]
  RETURNS [position: Position];
```

Clients may determine the position of the first character on any line by calling.

```
GetPosition: PROCEDURE [sw: Window.Handle, line: CARDINAL]
  RETURNS[Position];
```

Clients can position the top of a text subwindow to an arbitrary position within the text source by calling.

```
SetPosition: PROCEDURE [sw: Window.Handle, position: Position];
```

The following procedure first finds the next line break and then does a **SetPosition.**

```
PositionToLine: PROCEDURE [sw: Window.Handle, position: Position];
```

5.22.3 Information/Alteration Operations

The following operations allow clients to interrogate and alter internal aspects of a text subwindow.

When a text subwindow is moved or sized clients must call.

```
Adjust: ToolWindow.AdjustProcType;
```

Clients can enable/disable the blinking caret for an append or edit text subwindow by calling.

```
BlinkingCaret: PROCEDURE [sw: Window.Handle, state: OnOff];
```

All output to text subwindows is buffered for efficiency purposes. To ensure all pending output has really made it to the source clients can call.

```
ForceOutput: PROCEDURE [sw: Window.Handle];
```

The following two procedures allow clients to interrogate or alter the current options setting for a text subwindow.

```
GetOptions: PROCEDURE [sw: Window.Handle] RETURNS [options: Options];
SetOptions: PROCEDURE [sw: Window.Handle, options: Options];
```

The following two procedures allow clients to interrogate or alter the text source for a text subwindow.

```
GetSource: PROCEDURE [sw: Window.Handle] RETURNS [source: TextSource.Handle];
SetSource: PROCEDURE [
  sw: Window.Handle, source: TextSource.Handle, position: Position ← 0, reset: BOOLEAN ← TRUE];
```

The **reset** BOOLEAN indicates whether the display/source correspondance is valid or should be rebuilt.

The following procedures are used internally in building the menu and split view facilities. They are potentially useful for constructing client menu routines.

```
SplitView: PROCEDURE [sw: Window.Handle, key: Keys.KeyName, y: INTEGER];
```

Split a text subwindow **y** bits down from the top of **sw**. **key** should be **Yellow**.

```
DisplayHandleFromPlace: PROCEDURE [sw: Window.Handle, place: Window.Place]
  RETURNS [display: TextDisplay.Handle];
```

Returns the display object for the split that contains the **place**.

5.22.4 Activation Operations

Tajo provides facilties for notifying Tools that users are not interested in the display state of a particular Tool (i.e. make yourself tiny). The following procedures are provided to allow clients to make a similar request of a specific text subwindow.

```
Sleep: PROCEDURE [sw: Window.Handle];
```

Requests that the text subwindow package minimize it's storage and resource requirements by destroying all state related to the displaying of text.

```
Wakeup: PROCEDURE [sw: Window.Handle];
```

Requests that the text subwindow package recompute all it's display state.

5.22.5 Menu Operations

The following procedures are used by the text subwindow package to implement the **TextOps**

menu. They are presented here so clients may contruct their own menus that contain these operations. In the following descriptions a *display region* is either a subwindow or one of its splits.

FindMCR: Menu.MCRType;

Implements the **Find** command. Uses the current selection as the argument to find. If the current selection is contained in this display region then search from that position otherwise use the current top of this display region.

NormalizeInsertionMCR: Menu.MCRType;

Position the display region such that the line containing the insertion position is at the top of the display region.

NormalizeSelectionMCR: Menu.MCRType;

If is subwindow contains the current selection, position the display region such that the line containing the current selection is at the top.

PositionMCR: Menu.MCRType;

Convert the current selection as an octal number and position the display region such that the line containing that position is at the top of the display region.

SplitMCR: Menu.MCRType;

Splits the display region into two splits.

WrapMCR: Menu.MCRType;

Toggles the wrap boolean in the text subwindow options record.

5.23 Tool

Many Tool writers want the user interface mechanisms of Tajo without worrying about the details of invocation. The Tool interface reduces the client's needed knowledge of the more basic levels of Tajo to a minimum. Refer to the Simple and Sample Tool descriptions in the appendices for examples of Tools that uses the Tool interface.

5.23.1 Tool Creation

The first thing a Tool does is call:

```
Create: PROCEDURE [
 name: STRING,
 makeSWsProc: MakeSWsProc,
 initialState: State ← default,
 clientTransition: ToolWindow.TransitionProcType ← NIL,
 movableBoundaries: BOOLEAN ← TRUE,
 initialBox: Window.Box ← ToolWindow.nullBox]
 RETURNS [window: Window.Handle];
MakeSWsProc: TYPE = PROCEDURE [window: Window.Handle];
```

The **name** parameter is the string that appears in a Tool's black name band. From this string are derived the strings that are used in the tiny box and inactive menu.

When the **initialState** is **default** the Tool assumes a predetermined state depending on how it is created. The Tool is initialized to be inactive when loaded from the command line in order to facilitate building image and checkpoint files with collections of inactive Tools. The Tool is initialized to be active when loaded while the user is in Tajo because it is likely that he wants to use the Tool right away.

If the **clientTransition** procedure is not NIL it is called before the Tool is about to change state and before anything is done to the data managed by the Tool interface. The one exception to this ordering rule is that **FormSW.FreeAllItems** is called for each FormSW in the Tool when the Tool is going inactive before the client's transition procedure is called. This is done because it is common for a client's transition procedure to deallocate a record containing data that the **FreeAllItems** procedure references. Thus, the data must be referenced before it goes away. [If the client doesn't like being called in this order he could set his own procedure to be the window transition procedure which could call **Tool.Transition**.]

When the **movableBoundaries** parameter is TRUE the user may select the boundary line between subwindows and reposition it.

The **initialBox** parameter can be used to specify the Tool box (bitmap relative). A value of **ToolWindow.nullBox** lets Tajo assign the box from the next available box slot.

5.23.2 Subwindow Creation

At various points, depending on the initial state of the Tool and user actions, the **makeSWsProc** procedure supplied to **Create** is called by Tajo in order to give the client the opportunity to create subwindows and menus. In the **makeSWsProc** procedure, the client may call one or more of the following procedures to create subwindows:

```
MakeFileSW: PROCEDURE [
    window: Window.Handle,
    name: STRING,
    access: TextSW.Access ← append,
    h: INTEGER ← DefaultHeight]
    RETURNS [sw: Window.Handle];

MakeFormSW: PROCEDURE [
    window: Window.Handle,
    formProc: FormSW.ClientitemsProcType,
    h: INTEGER ← DefaultHeight]
    RETURNS [sw: Window.Handle];

MakeMsgSW: PROCEDURE [
    window: Window.Handle,
    lines: INTEGER ← 1]
    RETURNS [sw: Window.Handle];

MakeStringSW: PROCEDURE [
    window: Window.Handle,
    access: TextSW.Access ← append,
    h: INTEGER ← DefaultHeight]
    RETURNS [sw: Window.Handle];

MakeTTYSW: PROCEDURE [
    window: Window.Handle,
    name: STRING,
    h: INTEGER ← DefaultHeight]
    RETURNS [sw: Window.Handle];
```

Clients can use above procedures succesfully with only cursory knowledge of Tajo.

The client can use other methods to create subwindows and then communicate the existence of them to the Tool interface by calling.

```
AddThisSW: PROCEDURE [window, sw: Window.Handle, swType: SWType ← predefined];
```

[Warning: Usually the **Create** call hasn't returned when the **makeSWsProc** procedure is called. This means that the **Window.Handle** variable into which the client assigns the value returned from **Create** is uninitialized. Thus, the client should not reference this variable in his **makeSWsProc** procedure. Instead, the client should use the **window** parameter passed to the **makeSWsProc** procedure].

5.23.3 Unique SWTypes

The Tool interface can manage client defined subwindow types just as it manages the **predefined** subwindow types: FormSW, FileSW, MsgSW, StringSW and TTYSW.

```
SWType: TYPE = MACHINE DEPENDENT {vanilla(0), predefined(376B), last(377B)};
```

Defining your own subwindow type is covered in Sec. 4.3. To register a subwindow type with the Tool interface call:

```
RegisterSWType: PROCEDURE [
    adjust: ToolWindow.AdjustProcType ← SimpleAdjustProc,
    sleep: SWProc ← NopSleepProc,
    wakeup: SWProc ← NopWakeupProc]
    RETURNS [uniqueSWType: SWType];
```

The **adjust** procedure is called whenever the user causes the subwindow size to change or be moved. The **sleep** procedure is called whenever the window in which the subwindow lives becomes tiny. The subwindow is then expected to throw away any data that it uses only to display its contents. The **wakeup** procedure undoes what **sleep** did when the Tool becomes active again.

If the client wanted to register a subwindow type that would use the **SimpleAdjustProc**, the **NopSleepProc** and the **NopWakeupProc** he could instead refer to the subwindow as **vanilla.**

5.23.4 Destruction and Deallocation

Normally, anything that the client creates should be destroyed by him before a Tool goes inactive, e.g. any private data. The Tool mechanism relieves the client of the chore of destroying subwindows and menus that were created in a standard way. In particular, menus should be created by a call to **Menu.Make**, **FormSW.ItemDescriptors** should be created by a call to **FormSW.AllocateItemDescriptor**, **FormSW.ItemObjects** should be created by calls to **FormSW.*Item** procedures.

The following procedure has the side effect of calling the **clientTransition** procedure with a **new** state of inactive before the Tool is destroyed.

```
Destroy: PROCEDURE [window: Window.Handle];
```

5.23.5 Utilities

A Tool might want to switch one subwindow for another subwindow in a Tool. This can be done by a call to:

```
SwapSWs: PROCEDURE [
   window, oldSW, newSW: Window.Handle,
   newType: SWType ← predefined]
   RETURNS [oldType: SWType]
```

> **window** is the Tool window. **oldSW** identifies the subwindow that is currently displayed that will be replaced by **newSW**. **newSW** can not currently be part of the tree that makes up the hierarchy of displayed windows. When this procedure has returned **oldSW** has been removed from this tree. **Error[code: swNotFound]** may be raised from this procedure.

The following procedure guarrentees unique log file names among FileSWs and TTYSWs:

```
UnusedLogName: PROCEDURE [unused, root: STRING];
```

> **unused**'s length is set to 0, **root** is appended to **unused**, some designation of the running environment is appended to **unused** (nothing when in Tajo, 'D when in the debugger and 'I when in the internal debugger) and (if the **unused** is still not unique) a number is appended to **unused.**

5.23.6 Errors

```
ErrorCode: TYPE = {notATool, unknownSWType, swNotFound, invalidWindow};
Error: SIGNAL [code: ErrorCode];
```

Any procedure that takes a **window** argument can raise **invalidWindow** (if **window** is not valid) or **notATool** (if **window** was not created by **Create**). **unknownSWType** can be raised by any procedure that takes a **SWType** argument.

5.24 ToolDriver

The ToolDriver interface allows a Tool to inform the ToolDriver package of its existence, and of the existence of its subwindows. This allows the ToolDriver package to make use of the functions provided by a Tool on behalf of a user communicating with the package via a script file. Every Tool should use the ToolDriver facilities if it is providing some generally useful function. Although the ToolDriver is an add-on package (i.e. it is not built into the regular Tajo), the interface routines are available in Tajo even without the ToolDriver so that the Tool being STARTed need not concern itself with unbound procedures.

To announce its existence, a Tool should call

```
NoteSWs: PROCEDURE [tool: STRING, subwindows: AddressDescriptor];

AddressDescriptor: TYPE = DESCRIPTOR FOR ARRAY OF Address;

Address: TYPE = RECORD [name: STRING, sw: Window.Handle];
```

tool is whatever name the Tool wishes to go by for purposes of the ToolDriver. It need not be the same as the name displayed in the herald of the Tool's window, and in general it will be different because the ToolDriver imposes the restriction that **tool** contain only alphanumerics. **subwindows** is a list of subwindows that the Tool wishes to make available to the ToolDriver. The **name** for each of these must also obey the restriction to contain only alphanumerics.

When a Tool goes inactive, unless it is prepared to be called by the ToolDriver while inactive, it should call

```
RemoveSWs: PROCEDURE [tool: STRING];
```

Tools that register with the ToolDriver interface should have unique names in each of the menus used by the Tool so as not to be ambiguous to the ToolDriver package.

5.25 ToolFont

The following routines provide Tajo's interface to the more primitive Vista **WindowFont** facilities. Basically these routines provide font storage management and/or font swapping.

```
Create: PROCEDURE [Compatibility.FHandle] RETURNS [WindowFont.Handle];
```

Allocates a font object, initializes it, and provides routines to manage font swapping (if appropriate).

```
Destroy: PROCEDURE [WindowFont.Handle];
```

Destoys the data segment and font object.

```
StringWidth: PROCEDURE [string: STRING, font: WindowFont.Handle ← NIL]
 RETURNS [[0..LAST[INTEGER]]];
```

Computes the width of the passed **string**. This routine maps non-printing characters (e.g. control characters and etc.) into a font specific default character.

5.26 ToolWindow

The Window illusion is one of the central notions of the Tajo. The **ToolWindow** interface provides the functions required to implement the window illusion for Tajo and relies heavily on the interface **Window**, in the Vista window package.

Most useful TYPES are copied from Window into the ToolWindow interface.

5.26.1 Tajo's use of Windows

Vista defines a window object which is a record which contains only the information necessary to define and operate upon a window, independent of Tajo. Within this section of the specification we will discuss only the fields within the window object that are germane to the functions provided by ToolWindow.

Tajo implements a mechanism for associating Tool specific data with Vista window objects that is transparent to clients of Tajo. Such information is omitted from the following descriptions.

As mentioned earlier, Tajo specializes Vista's windows. The five defined types are:

```
WindowType: TYPE = {root, tool, clipping, sub, other};
```

> In descending levels of the window tree: the **root** window is the bitmap, a **tool** window is referred to in this document as a Tool window, a **clipping** window is associated with each **tool** window and should be of no concern to clients, **sub** windows are functional display areas and **other** windows are all lower levels.

In the event you have a pointer to a window and you want to find out what type of window it is you can call.

```
Type: PROCEDURE [window: Handle] RETURNS [WindowType];
```

A box is a rectangular part of something -- either a display screen or the bitmap or a window or a subwindow. Vista also does the appropriate clipping required when actually putting bits into the objects. The fields in a box's defining record are the x and y coordinates of its top left corner, relative to the window that it is a part of, and its width and height. Note that the box can extend "beyond" the edges of the thing that it is in.

```
Place: TYPE = RECORD [x, y: INTERGER];
Dims: TYPE = RECORD [w, h: INTEGER];
Box: TYPE = RECORD [place: Place, dims: Dims];
```

5.26.2 Tool Windows

Several routines are provided for the creation and adjustment of the tool window and all its children -- that is, of the space occupied by the Tool's window, as distinct from the content within the window.

These routines allow a Tool to create new windows. They also allow for the re-ordering of the stack of windows, thus altering which windows are *on top of and contain* others.

It is the Tajo philosophy that the (human) user of the Tools should be able to arrange the windows on his display any way he wants. More importantly, the Tool is at the mercy of the user in terms of the size and position of windows. Tools should be written with this understanding.

The following routine is provided for the creation of a tool window:

```
Create: PROCEDURE [
 name: STRING,
 adjust: AdjustProcType,
 transition: TransitionProcType,
 box: Box ← nullBox,
 limit: LimitProcType ← StandardLimitProc,
 initialState: State ← active,
 named: BOOLEAN ← TRUE]
 RETURNS [Handle];
```

This procedure creates a new **tool** window with the indicated box. A **nullBox** value is used to indicate automatic system allocation of initial position and size. If **named** is TRUE the window will have a black band across the top which displays **name**. The three procedures which are passed as arguments are discussed in detail below.

Tool windows can be in one of the following states.

```
State: TYPE = {inactive, tiny, active};
```

When a tool window is inactive this is an indication to the Tool that the user is not now interested in any of the functions implemented by this Tool and all resources utilized by the Tool should be freed. When the tool window is tiny this is an indication to the Tool that the user is not interested in the display and that all resources asscoiated with the display state should be freed.

The state of a Tool window is returned from

```
GetState: PROCEDURE [window: Handle] RETURNS [state: State];
```

Tool windows can have one of the following size attributes.

```
Size: TYPE = {tiny, normal, zoomed};
```

When a Tool is tiny a small rectangular box is displayed that contains some text derived from the name of the Tool. This text can be changed by a call to:

```
SetTinyName: PROCEDURE [window: Handle, name: STRING, name2: STRING ← NIL];
```

name is the first line of text and **name2** is the second.

When a Tool is inactive a menu entry whose text is derived from the name of Tool is placed on the *Inactive* menu. To change the Tool name call:

```
SetName: PROCEDURE [window: Handle, name: STRING];
```

The following two procedures return heap strings that describe the **STRING**s used as names in the various Tool states. The client must free these heap strings.

```
GetName, GetInactiveName: PROCEDURE [window: Handle] RETURNS [name: STRING];
GetTinyName: PROCEDURE [window: Handle] RETURNS [name, name2: STRING];
```

Three client-defined procedures are supplied on a create tool window call. The functions of these procedures are as follows:

5.26.2.1 Adjust and Limit Procedures

Although it is the user who, in general, moves windows around on the display in order to satisfy his own ideas about a suitable arrangement, Tajo allows the individual Tools to, a) know when one of their windows has been so adjusted and, b) exercise veto or modification rights over moves. The latter is particularly useful in allowing a Tool to prohibit, for example, its window becoming smaller than some certain size or being moved completly off the visible display region.

The Tool's limit procedure is of the form

```
LimitProcType: TYPE = PROCEDURE [window: Handle, box: Box] RETURNS [Box];
```

The Tool's adjust procedure is of the form

```
AdjustProcType: TYPE = PROCEDURE [window: Handle, box: Box, when: When];
When: TYPE = {before, after};
```

Whenever the system is about to adjust the window's location or size, it calls the limit procedure. It then uses the returned **box** to call the Tool's adjust procedure. The adjust procedure is called both before and after the actual adjustment is made. The adjust procedure is also called whenever the bitmap is altered. Unlike previous versions of Tajo it is now appropriate to manipulate subwindow boxes in the adjust procedure by procedure calls on the Window interface.

The following procedure performs what we consider the *normal* window limiting operations.

```
StandardLimitProc: LimitProcType;
```

5.26.2.2 Transition Procedure

The Tool is notified whenever a user action causes Tajo to change the state of the Tool.

```
TransitionProcType: TYPE = PROCEDURE [window: Handle, old, new: State];
```

A Tool's TransitionProc can be changed by a call to:

```
SetTransitionProc: PROCEDURE [window: Handle, proc: TransitionProcType]
 RETURNS [TransitionProcType];
```

After a Tool window is all set up, the client should call the following procedure that causes **window**, and its subtree of windows, to be displayed:

```
Show: PROCEDURE [window: Handle];
```

The following procedure removes **window** from the group of windows displayed on the bitmap:

```
Hide: PROCEDURE [window: Handle];
```

The following procedure is used to destroy both tool windows and subwindows.

```
Destroy: PROCEDURE [window: Handle];
```

5.26.3 Subwindows

To aid the Tool in the manipulation of the content of the window, the notion of the subwindow has been invented. A subwindow is a box (rectangle defined by an x,y and a width and height) *within* the clipping window. Within is in quotes there, because although the subwindow box is defined in terms of the clipping window -- that is its coordinates are subwindow-relative, it may extend "outside" of the actual window -- say if its "x" is negative or its height is greater than that of the window.

Subwindows are normally created by the client in order to simplify his window manipulations. For instance, he can create a subwindow and arrange it to be the left half of his window. Then, if he draws pictures in the subwindow, the pictures will be truncated by the system when they reach the right edge of his subwindow and won't overlay whatever is in the right half of the window.

The system provides three procedures to define subwindows:

```
CreateSubwindow: PROCEDURE [
 parent: Handle,
 display: DisplayProcType ← NIL,
 box: Box ← nullBox,
 boxesCount: BoxesCount ← one]
 RETURNS [Handle]
```

> Creates a new subwindow object with the indicated **box** within its (as yet unspecified) window and enlinks it into the **parent** window's chain of subwindows.

5.26.3.1 Display Procedure

The **display** procedure is a procedure of the form

```
DisplayProcType: TYPE = PROCEDURE [Handle];
```

This procedure is called whenever the content of the window needs to be refreshed onto the bitmap display. This can be required, for instance, if a window previously on top of this window is moved out of the way.

This version of Tajo supports and encourages both partial and incremental display updating. The Vista package documentation describes how a display procedure accomplishes partial repainting. Therein the **boxesCount** parameter is explained. For all Tajo supplied subwindow types, display procedures are automatically supplied at create time.

The following two procedures are not normally used, as subwindows are enlinked upon creation.

```
EnlinkSubwindow: PROCEDURE [parent, child, youngerSibling: Handle]
```

Links the subwindow into the chain in the indicated position.

```
DelinkSubwindow: PROCEDURE [child: Handle]
```

Removes the subwindow and it's children from the window structure.

5.26.4 Window Content Manipulation

The Vista window package provides a large number of procedures for putting light and dark bits into the window or moving them about. These procedures are only of interest to clients who wish to construct their own subwindow types and we refer them to the Vista documentation.

5.26.5 Utilities

Inorder to determine the Tool window of a subwindow call

```
WindowForSubwindow: PROCEDURE [sw: Handle] RETURNS [window: Handle];
```

To enumerate all the subwindow within a Tool window call

```
EnumerateSWs: PROCEDURE [window: Handle, proc: EnumerateSWProcType];
EnumerateSWProcType: PROCEDURE [window, sw: Window.Handle]
 RETURNS [done: BOOLEAN];
```

To active or deactivate a Tool window call

```
Activate, Deactivate: PROCEDURE [window: Handle];
```

To change the Size of a Tool window call

```
MakeSize: PROCEDURE [window: Handle, size: Size];
```

To change the size and position of a Tool window call

```
SetBox: PROCEDURE [window: Handle, box: Box];
```

5.26.6 Errors and Abnormal Conditions

The ToolWindow routines generate no SIGNALs as a result of improper use or specification. In general, the conditions that one might expect to generate a SIGNAL just do nothing. This is in line with the philosophy that the window is really just showing the user a piece of an infinite plane -- the Tool can (attempt to) put things anywhere on that plane; only the portion of the plane within the window is displayed to the user.

5.27 TTYSW

The teletype subwindow interface enables traditional teletype interaction with a user. Other Tajo user interaction facilities are based on the notification concept. Since many programs are already written using a teletype-like control structure the teletype subwindow is available to clients.

5.27.1 Creation/Destruction

A Teletype subwindow is created by a call to:

```
Create: PROCEDURE [
    sw: Window.Handle,
    backupFile: STRING,
    s: Stream ← NIL,
    newFile: BOOLEAN ← TRUE,
    options: TextSW.Options ← defaultOptions];

defaultOptions: TextSW.Options = [access: append, menu: TRUE, split: TRUE,
  wrap: TRUE, scrollbar: TRUE, flushTop: FALSE, flushBottom: FALSE];
```

The **backupFile** parameter specifies the name of the file on which the teletype subwindow writes. The string is copied. However, if **s** is not NIL then **s** is the stream handle on the file used. When **newFile** is TRUE the length of the file is set to zero at create time else the existing length is used.

Once the teletype subwindow is created the client must FORK a process to do Input. This process should be able to handle the following SIGNALS and ERRORS:

```
LineOverflow: SIGNAL [s: STRING] RETURNS [ns: STRING];
```

Indicates that input has filled the string **s**, the current contents of the string is passed as a parameter to the **SIGNAL**. The catch phrase should return a string **ns** with more room.

```
Rubout: SIGNAL;
```

Indicates that the DEL key was typed during **TTY.GetEditedString**.

Additionally these exception condition could arise:

```
ABORTED: ERROR;
```

Indicates that the Input process has been aborted.

```
String.InvalidNumber: ERROR;
```

Indicates that the user entered an invalid number in one of the number getting procedures.

```
ErrorCode: TYPE = {notATTYSW};
Error: SIGNAL [code: ErrorCode];
```

Indicates that a passed in subwindow is not a TTYSW.

The following procedure destroys teletype subwindow attributes of the subwindow. However, before this procedure is called the Input process should be aborted.

```
Destroy: PROCEDURE [sw: Window.Handle];
```

If the client wishes to destroy the teletype subwindow from within the Input process he should instead call

```
DestroyFromBackgroundProcess: PROCEDURE [sw: Window.Handle];
```

as he returns from the Input process.

5.27.2 Input and Output

From the Input process, these are the provided input routines:

```
GetChar: PROCEDURE [sw: Window.Handle] RETURNS [CHARACTER];
```

Returns the next character typed by the user.

The following procedure is used by the remaining input procedures to get input from the user while allowing him simple editing functions.

```
GetEditedString: PROCEDURE [
  sw: Window.Handle, s: STRING, t: PROCEDURE [CHARACTER] RETURNS [BOOLEAN]
  newstring: BOOLEAN]
  RETURNS [CHARACTER];
```

User input is appended to the string **s**. The user-supplied procedure **t** determines which character terminates the string; **t** should return **TRUE** if the character **c** passed to it should terminate the string. If the **BOOLEAN newstring** is **FALSE**, characters are simply appended to **s**. If **newstring** is **TRUE**, the first character of input plays a deciding role: if it is the **Ascii.ESC** character, the current contents of **s** are displayed then subsequent input character(s) are appended to **s** (note: the **ESC** is not appended to **s**); however, if it is not **ESC**, **s** is initialized to empty. The **SIGNAL TTY.LineOverflow** is raised if **s.maxlength** is reached. The following special characters are recognized on input (and are not appended to **s**):

DEL - raises the **SIGNAL TTY.Rubout**
↑A, ↑H (backspace) - delete the last character (sends ↑H)
↑W, ↑Q (backword) - delete the last word (sends multiple ↑H)
↑X - delete everything (sends multiple ↑H), **s** is set to empty
↑R - retype the line (sends CR, LF, then **s**)
↑V - quote the next character, used to input special characters

The returned character **c** is the character which terminated the string; **c** is not echoed nor included in the string.

The following **Get*** procedures all call **GetEditedString** passing TRUE for **newstring.**

```
GetString: PROCEDURE [
 sw: Window.Handle, s: STRING, t: PROCEDURE [CHARACTER] RETURNS [BOOLEAN]]
 RETURNS [CHARACTER];
```

The terminating character is echoed. No value is returned.

```
GetID: PROCEDURE [sw: Window.Handle, s: STRING];
```

Input is terminated with a space or carriage return. The terminating character is not echoed.

```
GetLine: PROCEDURE [sw: Window.Handle, s: STRING];
```

Input is terminated with a carriage return.

These are numerial input routines:

```
GetNumber: PROCEDURE [sw: Window.Handle, default: UNSPECIFIED, radix: CARDINAL]
   RETURNS [UNSPECIFIED];
```

GetID followed by a call to **String.StringToNumber.** The value **default** will be displayed if **ESC** is type. **radix** is a default value, use the 'B or 'D notation to force octal or decimal. **radix** values. other that 8 or 10 cause unpredictable results.

```
GetDecimal: PROCEDURE [sw: Window.Handle] RETURNS [INTEGER];
```

GetID followed by a call to **String.StringToDecimal.**

```
GetOctal: PROCEDURE [sw: Window.Handle] RETURNS [UNSPECIFIED];
```

GetID followed by a call to **String.StringToOctal.**

Long numerial input routines are available:

```
GetLongDecimal: PROCEDURE [sw: Window.Handle] RETURNS [LONG INTEGER];
GetLongNumber: PROCEDURE [sw: Window.Handle, default: LONG UNSPECIFIED, radix: CARDINAL]
   RETURNS [LONG UNSPECIFIED];
GetLongOctal: PROCEDURE [sw: Window.Handle] RETURNS [LONG UNSPECIFIED];
```

The Put interface can be used to produce formatted output to the subwindow. However, clients can use the following output procedures:

```
AppendChar: PROCEDURE [sw: Window.Handle, char: CHARACTER];
AppendString: UserInput.StringProcType;
```

5.27.3 Utilities

The following procedure returns **TRUE** when the next character of output would start a new line:

```
NewLine: PROCEDURE [sw: Window.Handle] RETURNS [BOOLEAN];
```

The following procedure sets the echoing mode of characters entered using **GetEditedString**. It returns the previous state of the echoing mode. The default echoing mode is **TRUE**:

```
SetEcho: PROCEDURE [sw: Window.Handle, new: BOOLEAN] RETURNS [old: BOOLEAN];
```

5.28 UserInput

The **UserInput** interface provides the client with the routines that are used to interpret user actions. Note that this interface depends heavily upon the definitions type and data definitions contained in Keys interface.

5.28.1 Notification

Clients who are only interested in normal keyboard character input should skip this section and go directly to section 5.28.3 User TypeIn.

A Tool can get notified of a change in the user state at the Interrupt Level and the Processing Level. Interrupt Level notifications usually take place within one vertical retrace of when the (hardware) stimulus occurs. Processing Level notifications occur in the order in which the stimuli occured, but may be far removed from the stimulus in time.

Processing Level

The Processing Level is where the usual processing of user input happens. The basic notion is embodied in the phrase "Don't call us, we'll call you". The real *interface* here is a set of Processing Notification Routines, or PNRs, one for each, Tajo defined, interesting device or event that may change state.

Here are the relevant type declarations for PNRs:

```
CursorPNRType: TYPE = PROCEDURE [window: Window.Handle, enterExit: EnterExit];
KeyPNRType: TYPE = PROCEDURE [
    key: Keys.KeyName, downUp: Keys.DownUp, window: Window.Handle, place:
    Window.Place];
EnterExit: TYPE = {enter, exit};
KeyPNRClass: TYPE = {keyset, keyboard, redButton, yellowButton, blueButton}

KeyName: TYPE = Keys.KeyName;
Button: TYPE = KeyName[Red..Yellow];
Key: TYPE = KeyName[Five..R8];
Paddle: TYPE = KeyName[Keyset1..Keyset5];
```

We stated earlier that there is a problem in deciding which Tool should receive the notification for a particular user action. Tools are notified of all user actions that occur when the cursor is in the Tool's window(s) or more specifically a subwindow within the Tool's window. By associating a context with each window object that contains the set of PNRs specific to that window, we can simply call the PNR associated with the specific user action.

> Fine Point: The cursor is considered to be in only one window at a time. This is because windows that are "on top of" other windows are considered to "obscure" those that are "below" them.

Note that Tool writers are free to use the same PNR for more than one action. Also, the same PNR may be used in more than one window.

Various Tajo packages and Tools implement a wide range of useful PNRs that Tool writers should feel free (indeed, encouraged) to use. To set the PNRs for a window, the following procedures are available.

```
SetCursorPNR: PROCEDURE [
  window: Window.Handle, proc: CursorPNRType ← NIL];
```

Sets the cursor PNR for the indicated window to be the provided procedure. The procedure **proc** will be called whenever the subwindow is entered or exited. If **proc** is not supplied then the system default will be used.

```
SetKeyPNR: PROCEDURE [window: Window.Handle, keyClass: KeyPNRClass, proc: KeyPNRType
← NIL];
```

Sets the indicated key PNR for the indicated window to be the provided procedure. This procedure will be called whenever the key(s) changes state. If **proc** is not supplied then the system default will be used.

A standard cursor PNR is

```
NopCursorPNR: CursorPnrType;
```

A call on this PNR is always a Nop.

Three Tajo supplied key PNRs are

```
DefaultKeyPNR: KeyPNRType;
```

If this PNR is called the actions are indirected to the default window:

```
GetDefaultWindow: PROCEDURE RETURNS [Window.Handle];
```

```
IgnoreKeyPNR: KeyPNRType;
```

If the PNR is called with **downUp** equal **down**, it will blink the display, otherwise the call is a Nop.

```
NopKeyPNR: KeyPNRType;
```

A call on this PNR is always a Nop.

To set all the key PNRs for a window to be **DefaultKeyPNR**, **IgnoreKeyPNR** or **NopKeyPNR**, call

```
SetDefaultPNRs: PROCEDURE [window: Window.Handle];
SetIgnorePNRs: PROCEDURE [window: Window.Handle];
SetNopPNRs: PROCEDURE [window: Window.Handle];
```

respectively. All of these procedures will set the cursor PNR to be **NopCursorPNR.**

The following procedures allow you to access useful data that is maintained by the Processing Level.

```
GetCurrentCursorPosition: PROCEDURE [Window.Handle] RETURNS [Window.Place];
```

Returns the "current" state of the Processing Level cursor position (the coordinates of the "hot spot" of the cursor in the specified windows coordinates), i.e., the state as of the last Processing Level notification (the last call on a PNR).

```
EnumerateDownKeys: PROCEDURE
    [window: Window.Handle, downUp: DownUp, place: Window.Place];
```

This procedure will generate a call on the appropriate PNR for *all* keys that are **downUp.** This means when you call this procedure it will process the current hardware state and call the appropriate PNR, in the window, for every key that is in the same state as the passed **downUp.**

Interrupt Level

We do not envision that clients will really be concerned with the Interrupt Level. We present this interface primarily for understanding purposes and for the rare cases that will require it.

User Input maintains two procedure variables that contain descriptors for the two Interrupt Notification Routines (SNRs). Here are the type declarations for the two SNRs:

```
MouseSNRType: TYPE = PROCEDURE RETURNS [screenPlace: Window.Place];
```

The primary purpose of this procedure is to track the mouse.

```
KeySNRType: TYPE = PROCEDURE [
    key: KeyName, du: DownUp, screenPlace: Window.Place];
```

The mouse SNR is called every time the Interrupt Level gets control of the cpu. The key SNR is called only if a paddle, button, or key has changed state. The Interrupt Level has routines to alter the procedure variables so that a tool can get its own SNRs invoked.

The two SNR variables are initialized to point at the system supplied ones. The default mouse SNR merely sees that the cursor doesn't go off the screen. The default key SNR causes items to be enqueued for later processing by the Processing Level. There is a separate call on the key SNR for each depression or release of a key even if more than one happens "simultaneously". [Note that no items are enqueued for subwindow boundary crossings. Queue items pertain only to a **DownUp** of a Key.]

SwapMouseSNR: PROCEDURE [new: MouseSNRType] RETURNS [old: MouseSNRType];

This procedure allows you to subsitute your own mouse SNR for the current one.

SwapKeySNR: PROCEDURE [new: KeySnrType] RETURNS [old: KeySnrType];

These procedures allow you to subsitute your own mouse or key SNR for the current one.

We only envision use of **SwapMouseSNR** or **SwapKeySNR** in atypical situations. Considerable knowledge is required to do your own processing at the Interrupt Level. If it's really necessary, you should use the system SNR's as prototypes.

The following procedures are used by the Interrupt and Processing levels. They should only be of interest if you are writing your own stimulus level routines (SNRs).

DequeueUA: PROCEDURE RETURNS [key: Keys.KeyName, downUp: DownUp, screenPlace: Window, action: BOOLEAN];

Removes (dequeues) a user action and returns the head item of the queue. If the queue is empty, **action** is FALSE.

EndOfUAQ: PROCEDURE RETURNS [BOOLEAN];

Returns TRUE if there are no items (user actions) in the queue, FALSE otherwise.

EnqueueUA: PROCEDURE [key: Keys.KeyName, downUp: DownUp, screenPlace: Window.Place];

Enqueues a user action (item) in the queue.

FlushUAQ: PROCEDURE;

Sets the user action queue empty ignoring any items in the queue.

5.28.2 Character Translation

The Alto hardware presents an *unencoded* bit interface for the keyset and keyboard. Tajo contains tables and procedures for doing key stroke to ASCII character translation using the definitions and notification mechanism described above.

The following procedure translates keyboard and/or keyset actions into characters

TranslateKeyIntoChar: PROCEDURE [key: Key] RETURNS[char: CHARACTER];

5.28.3 User TypeIn

The User TypeIn facilities are built using the above described notification and character translation facilities. The TypeIn facility lets the client supply a procedure that will be called whenever the appropriate actions have taken place that correspond to a character being typed (e.g. key down, key

up, shift, control etc.). TypeIn allows the client to essentially be free of any concern for "how it is done". There is one important exception to this rule: there can be changes made in a subwindow that are not noticable above the TypeInPNR level. If a Tool wishes to see these changes, it must operate at or below the TypeInPNR level.

Clients of the type in mechanism supply procedures of the following TYPE.

```
CaretProcType: TYPE = PROCEDURE [window: Window.Handle, startStop: StartStop];

StringInProcType: TYPE = PROCEDURE [window: Window.Handle, string: STRING];
```

These two procedures allow clients to create/destroy type in for a specific subwindow.

```
CreateStringInOut: PROCEDURE [
    window: Window.Handle,
    in, out: StringProcType,
    caretProc: CaretProcType ← NopCaretProc];
```

Error[code: windowAlreadyHasStringInOut] can be raised if the window has type in.

```
DestroyStringInOut: PROCEDURE [window: Window.Handle];
```

The following procedures allow clients to alter the procedures to be called for a window with already existing type in.

```
SetStringIn: PROCEDURE [
    window: Window.Handle, proc: StringProcType];
    RETURNS[StringProcType];

SetStringOut: PROCEDURE [
    window: Window.Handle, proc: StringProcType];
    RETURNS[StringProcType];
```

For the above two procedures **Error[code: noStringInOutForWindow]** can be raised if the window has no type in.

The typein mechanism is also designed to allow a client to redirect input/output to another window (Tool or subwindow). For example, a Tool has a MsgSW and a FileSW which accepts user type-in. Type-in to the MsgSW could be redirected to the FileSW so that the user would only have to have the cursor in the Tool window and not specifically in the FileSW when typing to the FileSW.

```
CreateIndirectStringInOut: PROCEDURE [from, to: Window.Handle];
```

Error[code: windowAlreadyHasStringInOut] can be raised if the window has type in.

```
DestroyIndirectStringInOut: PROCEDURE [window: Window.Handle];
```

The following procedures allow clients to drive the type in mechanism as though the data was coming from the user. The returned BOOLEANs are TRUE only if **window** was prepared to accept input.

```
StuffCharacter: PROCEDURE [window: Window.Handle, char: CHARACTER];
```

```
    RETURNS[BOOLEAN];
StuffCurrentSelection: PROCEDURE [window: Window.Handle];
    RETURNS[BOOLEAN];
StuffString: PROCEDURE [window: Window.Handle, string: STRING];
    RETURNS[BOOLEAN];
```

The following procedure allows clients to output directly to a window, bypassing any input filtering that might have been performed.

```
StringOut: PROCEDURE [window: Window.Handle, string: STRING];
```

5.28.4 Utilities

Client operations that run for more than a few seconds can poll

```
userAbort: READONLY BOOLEAN;
```

to see if the user has indicated that he wants to abort the operation by keying some abort sequence. Everytime that a PNR is called this variable is set to FALSE. In the unusual case that a client needs to explicitely set this variable to FALSE he should call

```
ResetUserAbort: PROCEDURE;
```

Sometimes a client is deep in the call stack of some notifier invoked operation from which he simply wants to UNWIND. The following ERROR can be raised that will be caught at the top level of the PNR mechanism.

```
ReturnToNotifier: ERROR [string: STRING];
```

> The client can catch the ERROR, post a message with **string** in it and let the ERROR propagate on up.

A standard Tajo mechanism of prompting for user confirmation is to set the cursor to **mouseRed** and call

```
WaitForConfirmation: PROCEDURE RETURNS [place: Window.Place, okay: BOOLEAN];
```

> If **okay** = TRUE then the user pushed the red mouse button otherwise the user pushed either the yellow or the blue mouse buttons. **place** is the position of the cursor, bitmap relative, when the button went down.

The cursor should be set back to its previous type upon return from the above procedure.

The following procedure returns when all the mouse buttons are released.

```
WaitNoButtons: PROCEDURE;
```

Clients sometimes want to wakeup at regular time intervals to do some operation. Mesa's condition variable timeout mechanism can be used to do this. However, sometimes a client needs to do a series of operations that if done while the the PNR mechanism was invoking some other operation either

would preempt the user or could cause serious problems in Tajo, e.g., blinking the cursor. Thus, the *periodic notification* mechanism is provided.

```
PeriodicNotifyHandle: TYPE = POINTER TO PeriodicNotifyEntry;
PeriodicNotifyEntry: TYPE;
PeriodicProcType: TYPE = PROCEDURE [window: Window.Handle, place: Window.Place];

CreatePeriodicNotify: PROCEDURE [
 proc: PeriodicProcType, window: Window.Handle, rate: Process.Ticks]
 RETURNS [PeriodicNotifyHandle];
```

> **proc** is called every interval defined by **rate** as long as no other PNR operations are taking place.

```
CancelPeriodicNotify: PROCEDURE [PeriodicNotifyHandle]
 RETURNS [nil: PeriodicNotifyHandle];
```

> Stops the periodic notification. Raises **Error[code: noSuchPeriodicNotifier]** if the passed in handle is not valid (NIL is a no-op.

5.29 UserTerminal

The interface **UserTerminal** describes the state of the user input/output devices (i.e. display bitmap, display cursor, keyboard, mouse, and keyset), and allows the client to manipulate them. This interface takes as fixed many of the characteristics of these devices and only allows variations such as the number of keys or the size and resolution of the display. This interface deals with many of the lowest level attributes of the terminal and, with a few exceptions, should not be of interest to Tajo clients. This section only presents definitions and functions of general interest to the Tajo client. The Vista documentation describes other operations.

[Warning: Do not call UserTerminal.SetState. Instead call TajoMisc.SetState].

Clients can determine the physical attributes of the display via the following exported variables.

```
screenWidth: READONLY CARDINAL[0..32767];
screenHeight: READONLY CARDINAL[0..32767];
pixelsPerInch: READONLY CARDINAL;
```

The bitmap display is addressed by xy coordinates defined as follows.

```
Coordinate: TYPE = MACHINE DEPENDENT RECORD [x, y: INTEGER];
```

The state of the display is defined as:

```
State: TYPE = {on, off, disconnected};
```

on - The display is physically on and visible to the user (bitmap allocated).

off - The display is physically off and not visible to the user (bitmap allocated).

disconnected - The same as **off** with no allocated bitmap.

Clients may determine the current state of the bitmap display by calling

```
GetState: PROCEDURE RETURNS [state: State];
```

The bitmap display is capable of displaying black-on-white or white-on-black. Clients may determine or alter the current state of the background by using the following procedures.

```
GetBackground: PROCEDURE RETURNS [background: Background];
SetBackground: PROCEDURE [new: Background] RETURNS [old: Background];

Background: TYPE = {white, black};
```

Clients may momentarily *blink* (video reverse) the display by calling

```
BlinkDisplay: PROCEDURE;
```

5.30 Window

This interface is only of interest to clients who are implementing their own subwindow types. It is also the primary interface for the Vista window package and is documented as a part of that package. The following occur elsewhere in this document and are included here to reduce the number of levels of indirection needed to understand Tajo.

```
Handle: TYPE = POINTER TO Object;
Object: TYPE = RECORD [...];
```

5.31 WindowFont

The following description of the interface **WindowFont** is taken from the Vista documentation. It is included in this document as convenience to clients. Vista font routines deal only in .strike fonts. Vista provides routines for initializing and manipulating fonts and font objects. For details of these operations we refer you to the Vista documentation.

The text painting procedures of the **Window** interface take as an argument a **Handle** on an object from **WindowFont**. The fields of a **Handle** are mostly private to the implementation.

```
Handle: TYPE = POINTER TO Object;
Object: TYPE = RECORD [...];
```

The bits within the font object that define the character pictures are private to the implementation. The only public interfaces allow the client to determine the sizes of the characters in screen dots:

```
CharWidth: PROCEDURE [char: CHARACTER, font: Handle ← NIL] RETURNS [[0..LAST[INTEGER]]]
```

```
FontHeight: PROCEDURE [font: Handle ← NIL] RETURNS [[0..LAST[INTEGER]]]
```

A **font** argument of NIL for these routines, as well as for the text painting routines of the **Window** interface, means to use the **defaultFont**. The **defaultFont** is set by calling

```
SetDefault: PROCEDURE [font: Handle]
```

Using this defaulting mechanism before the **defaultFont** is set is a client error.

6.0 OPERATIONAL CONSIDERATIONS

The Compatiblity interface contains useful types that enable source level compatiblity between Tools written for AltoMesa Tajo and Pilot Tajo.

6.1 AltoMesa version

The following procedures apply to the Alto world and should be called instead of their counter parts in ImageDefs:

```
TajoMisc.MakeCheckPoint: PROCEDURE [name: STRING];
TajoMisc.MakeImage: PROCEDURE [name: STRING];
```

6.2 Pilot version

Code that works on AltoMesa may not work on Pilot due to more extensive monitoring of Tajo and the preemptive process mechanism in Pilot.

GLOSSARY

button: One of the three (sometimes 2) things on a mouse that go up and down.

choice: The process of pointing at a portion of the screen with the mouse and clicking a button such that some operation is performed. (See select)

contents: The file or data that the Librarian Data Base is keeping track of.

current selection: The system global selection. The argument to some menu commands.

form: A collection of values that the user may alter that usually serve as parameters to some operaion. Commands may be invoked directly from forms.

Interrupt Level: The process responsible for capturing user actions and enqueuing them for subsequent processing.

Interrupt Notification Routine (SNR): A procedure that may be notified on the Interrupt Level.

key: One of the things on a keyboard that go up and down.

Libject: Shorthand for **Li**brarian Ob**ject**. The name of an item contained in the Librarian Data Base.

Librarian Data Base: The complete history data files and data stored and controlled by the Librarian Service.

Librarian Interface: The mechanism used to access the data stored in the Librarian Data Base.

Librarian Service: The network based program that controls the Librarian Data Base.

menu: A list of options and commands presented to the user that is displayed due to a button depression. A choice can be made from this list.

Menu Command Routine (MCR): A procedure that runs as a result of a menu item being chosen.

notification mechanism (the notifier): The mechanism whereby PNRs and SNRs get notified of user actions.

notify: Invoke a PNR or SNR; call a PNR or SNR to notify it that a user action has occurred.

paddle: One of the five things on a keyset that go up and down.

Processing Notification Routine (PNR): A procedure that may be notified on the Processing Level.

Processing Level: The process that implements the notification mechanism.

PropertyList: The data structure used for passing as well as receiving data from the librarian database.

selection: The process of pointing at text or graphics on the screen with the mouse and clicking a button such that it becomes highlighted in some way. Also, the data so selected.

SNR: See *Interrupt Notification Routine* definition.

subwindow: A rectangular sub-region of a Tool.

Tool: A program that runs *in* Tajo, but is not *part* of it.

Tajo: The basic runtime system for tools.

user action: The depression or release of a paddle, button or key by a human user. Sometimes the term includes the moving of the cursor across a subwindow boundary.

Vista: General window management software package.

window: A rectangular region on the display. The primary output medium for a Tool.

Appendix 1: A Simple Tool

```
-- File: SimpleTool.mesa - last edit by:
-- Mark, Sep 22, 1980 2:19 PM

-- This is an example of a minimal "Tool" that runs in Tajo.  It is the equivalent of
-- everyone's first Mesa program to "read a character and echo it on the display".
-- We go beyond that to present a little of the Tajo religion.  It is our goal that when
-- a Tool is inactive it should consume minimal resources.

DIRECTORY
 Tool USING [Create, MakeFileSW, MakeSWsProc],
 Storage USING [FreeNodeNil, Node],
 ToolWindow USING [TransitionProcType],
 UserInput USING [SetStringIn, StringProcType],
 Window USING [Handle];

SimpleTool: PROGRAM IMPORTS Storage, Tool, UserInput =
 BEGIN

 -- TYPEs

 DataHandle: TYPE = POINTER TO Data;
 Data: TYPE = RECORD [
  -- File subwindow stuff
  fileSW: Window.Handle ← NIL,
  oldStringIn: UserInput.StringProcType ← NIL];

 -- Variable declarations

 -- This data illustrates a technique for minimizing memory use when this Tool is inactive
 toolData: DataHandle ← NIL;
 wh: Window.Handle; -- Tool's window

 -- Tool needed routines

 ClientTransition: ToolWindow.TransitionProcType =
  -- This procedure is called whenever the system determines that this
  -- Tool's state is undergoing a user invoked transition.
  -- In this example we minimize the memory requirements when we are inactive.
  BEGIN
  SELECT TRUE FROM
   old = inactive =>
    IF toolData = NIL THEN
     BEGIN toolData ← Storage.Node[SIZE[Data]]; toolData↑ ← []; END;
   new = inactive =>
    IF toolData # NIL THEN
     BEGIN toolData ← Storage.FreeNodeNil[toolData]; END;
   ENDCASE;
  END;

 Init: PROCEDURE =
  BEGIN
  wh ← Tool.Create[
```

```
    makeSWsProc: MakeSWs, initialState: default,
    clientTransition: ClientTransition, name: "Simple Tool 6.0"L];
  END;

MakeSWs: Tool.MakeSWsProc =
  BEGIN
  toolData.fileSW ← Tool.MakeFileSW[window: window, name: "Simple.log"L];
  -- Here we demonstrate a common augmentation trick used by Tajo clients.  We
  -- interpose our procedure between the notification mechanism and the file
  -- subwindow so that we can see what the user typed.
  toolData.oldStringIn ← UserInput.SetStringIn[toolData.fileSW, MyStringProc];
  END;

MyStringProc: UserInput.StringProcType =
  BEGIN
  -- To be useful the client would normally look at the characters as they go by.
  -- We just pass them on.
  toolData.oldStringIn[window, string];
  END;

-- Mainline code

Init[]; -- this gets string out of global frame

END...
```

Appendix 2: A Sample Tool

```
-- File: SampleTool.mesa - last edit by:
-- Mark, Sep 23, 1980 5:21 PM
-- Smokey, May 2, 1980 6:12 PM
-- Evans, Jul 10, 1980 12:43 PM

-- This is an example of a "Tool" that runs in Tajo. It demonstrates the use of a
-- comprehensive set of commonly used Tajo facilities. Specifically we present examples
-- of the definition, creation, use and destruction of the following:
--  Windows and subwindows
--  Menus
--  Msg subwindows
--  Form subwindows
--  File subwindows

DIRECTORY
 Ascii USING [CR],
 Menu USING [Handle, Instantiate, Make, MCRType],
 FormSW USING [
  AllocateItemDescriptor, BooleanChoices, BooleanItem, ClientItemsProcType,
  CommandItem, Enumerated, EnumeratedItem, line0, line1, line2, line3, line4,
  NotifyProcType, ProcType, StringItem],
 Put USING [Line],
 Tool USING [
  Create, MakeFileSW, MakeFormSW, MakeMsgSW, MakeSWsProc, UnusedLogName],
 Storage USING [CopyString, FreeNodeNil, Node],
 ToolWindow USING [TransitionProcType],
 Window USING [Handle];

SampleTool: PROGRAM IMPORTS FormSW, Menu, Put, Storage, Tool =
 BEGIN

 -- TYPEs

 StringNames: TYPE = {vanilla, password, readOnly};

 DataHandle: TYPE = POINTER TO Data;
 Data: TYPE = MACHINE DEPENDENT RECORD [
  -- Message subwindow stuff
  msgSW(0): Window.Handle ← NIL,
  -- File subwindow stuff
  fileSW(1): Window.Handle ← NIL,
  -- Form subwindow stuff
  -- Note: enumerateds and booleans must be word boundary
  -- aligned as addresses for them must be generated
  formSW(2): Window.Handle ← NIL,
  switch1(3): BOOLEAN ← NULL,
  switch2(4): BOOLEAN ← NULL,
  enum1(5): Enum1 ← NULL,
  enum2(6): Enum2 ← NULL,
  strings(7): ARRAY StringNames OF STRING ← NULL];

 Enum1: TYPE = {a, b, c};
```

```
Enum2: TYPE = {x, y, z};

-- Variable declarations

-- This data illustrates a technique for minimizing memory use when this Tool is inactive
toolData: DataHandle ← NIL;
wh: Window.Handle; -- Tool's window

-- Sample Tool Menu support routines

MenuCommandRoutine: Menu.MCRType =
 -- Do the tasks necessary to execute a menu command.
 -- If the command will take a long time, then one might FORK a PROCESS to do it.
 BEGIN
 SELECT index FROM
  0 => Put.Line[toolData.msgSW, "Message posted."L];
  1 => Put.Line[toolData.fileSW, "A Menu command called."L];
  ENDCASE => Put.Line[toolData.fileSW, "B Menu command called."L];
 END;

-- Sample Tool FormSW support routines

FormSWCommandRoutine: FormSW.ProcType =
 -- Do the tasks necessary to execute a form subwindow command.
 -- Again, if the command will take a long time, then one might FORK a PROCESS to do it.
 BEGIN
 Put.Line[toolData.fileSW, "The Command Procedure has been called."L];
 END;

NotifyClientOfFormAction: FormSW.NotifyProcType =
 -- This procedure will be called whenever a potentially interesting state
 -- change (user action) occurs in the Form subwindow.
 BEGIN
 Put.Line[toolData.fileSW, "The Notify Procedure has been called."L];
 END;

-- Tool needed routines

ClientTransition: ToolWindow.TransitionProcType =
 -- This procedure is called whenever the system determines that this
 -- Tool's state is undergoing a user invoked transition.
 -- In this Example we demonstrate a technique that minimizes the memory
 -- requirements for a Tool that is inactive.
 BEGIN
 SELECT TRUE FROM
  old = inactive =>
   IF toolData = NIL THEN
    BEGIN toolData ← Storage.Node[SIZE[Data]]; toolData↑ ← []; END;
  new = inactive =>
   IF toolData # NIL THEN
    BEGIN toolData ← Storage.FreeNodeNil[toolData]; END;
  ENDCASE;
 END;

Init: PROCEDURE =
 BEGIN
 wh ← Tool.Create[
  makeSWsProc: MakeSWs, initialState: default,
  clientTransition: ClientTransition, name: "Sample Tool 6.0"L];
```

```
  END;

MakeForm: FormSW.ClientItemsProcType =
  BEGIN OPEN FormSW; -- This procedure creates a sample FormSW.
  nItems: CARDINAL = 8;
  e1: ARRAY [0..3) OF Enumerated ←
   [["A"L, Enum1[a]], ["B"L, Enum1[b]], ["C"L, Enum1[c]]];
  e2: ARRAY [0..3) OF Enumerated ←
   [["X"L, Enum2[x]], ["Y"L, Enum2[y]], ["Z"L, Enum2[z]]];
  items ← AllocateItemDescriptor[nItems];
  toolData.strings[vanilla] ← toolData.strings[password] ← NIL;
  toolData.strings[readOnly] ← Storage.CopyString["Read Only String"L];
  -- Create an example of command item usage
  items[0] ← CommandItem[
   tag: "Command"L, place: [0, line0], proc: FormSWCommandRoutine];
  -- Create three examples of string item usage
  items[1] ← StringItem[
   tag: "Vanilla"L, place: [200, line0], string: @toolData.strings[vanilla],
   inHeap: TRUE];
  items[2] ← StringItem[
   tag: "Password"L, place: [0, line1], string: @toolData.strings[password],
   feedback: password, inHeap: TRUE];
  items[3] ← StringItem[
   tag: "ReadOnly"L, place: [0, line2], string: @toolData.strings[readOnly],
   readOnly: TRUE];
  -- Create two examples of apparent booleans
  -- The first one is actually done via an enumerated item
  items[4] ← EnumeratedItem[
   tag: "boolean(trueFalse)"L, place: [0, line3], feedback: all,
   value: @toolData.switch1, copyChoices: FALSE, choices: BooleanChoices[]];
  toolData.switch1 ← TRUE;
  items[5] ← BooleanItem[
   tag: "boolean(video)"L, place: [250, line3], switch: @toolData.switch2];
  toolData.switch2 ← TRUE;
  -- Create two examples of enumerated FormSWItem usage
  items[6] ← EnumeratedItem[
   tag: "enumerated(one)"L, place: [0, line4], feedback: one,
   value: @toolData.enum1, choices: DESCRIPTOR[e1]];
  toolData.enum1 ← a;
  items[7] ← EnumeratedItem[
   tag: "enumerated(all)"L, place: [175, line4], feedback: all,
   value: @toolData.enum2, choices: DESCRIPTOR[e2]];
  toolData.enum2 ← y;
  RETURN[items: items, freeDesc: TRUE];
  END;

MakeSWs: Tool.MakeSWsProc =
  BEGIN
  logName: STRING ← [40];
  menuStrings: ARRAY [0..3) OF STRING ←
   ["Post message"L, "A Command"L, "B Command"L];
  menu: Menu.Handle ← Menu.Make[
   name: "Tests"L, strings: DESCRIPTOR[menuStrings],
   mcrProc: MenuCommandRoutine];
  Tool.UnusedLogName[unused: logName, root: "Sample.log"L];
  toolData.msgSW ← Tool.MakeMsgSW[window: window];
  toolData.formSW ← Tool.MakeFormSW[window: window, formProc: MakeForm];
  toolData.fileSW ← Tool.MakeFileSW[window: window, name: logName];
  Menu.Instantiate[menu, window];
```

```
 END;

-- Mainline code

Init[]; -- this gets string out of global frame

END...
```