NISTIR 4471

THE TROI [TELEROBOTIC OPERATOR INTERFACE] USER'S GUIDE

Barry Warsaw
John Michaloski

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Intelligent Controls Group
Bldg. 220 Rm. B127
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4471

THE TROI [TELEROBOTIC OPERATOR INTERFACE] USER'S GUIDE

Barry Warsaw
John Michaloski

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Intelligent Controls Group
Bldg. 220 Rm. B127
Gaithersburg, MD 20899

January 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

Contents

1 Introduction

The TeleRobotic Operator Interface (*TROI*) is a robotic operator/programmer/human/user interface toolkit for the NASA Space Station Flight Telerobotic Servicer (*FTS*). The NASA/NBS Standard Reference Model for Telerobotic Control System Architecture (*NASREM*) outlines the responsibilities of the Space Station Telerobot operator interface (*OI*). NASREM states that the "operator interface provides a means by which the human operator, either in the space station or on the ground, can observe, supervise, and directly control the telerobot." In order to achieve this control, NASREM states that the operator and programmer interface provide the services to control, observe, define goals, indicate objects, edit both programs and data. NASREM defines a hierarchical architecture which requires that the OI share control at various levels of the hierarchy. TROI is a software toolkit that facilitates implementation of NASREM operator interfaces. TROI supplies a generic approach to operator-interfaces emphasizing graphical window-system based technology. This generic flexibility provides the ability to design and build operator-interfaces for all levels in the NASREM hierarchy. In real-time, TROI connects an X terminal based windowing environment via host workstation to the target Robot Control System (*RCS*). RCS is a real-time, multi-processor, shared-memory system supporting the NASREM architecture. Figure 1 shows a block diagram of the TROI architecture.

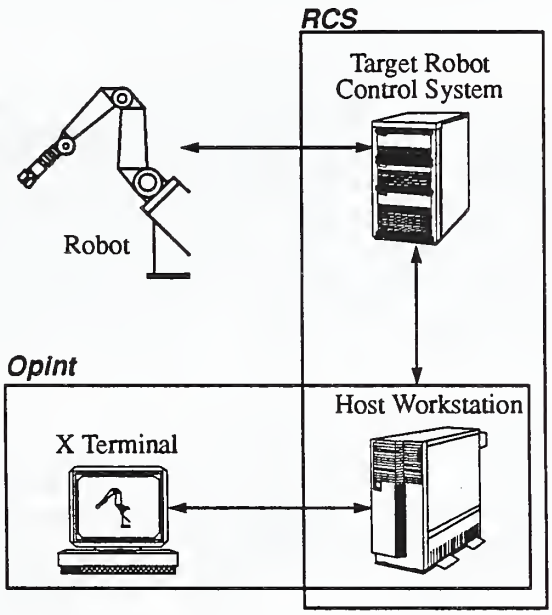

Figure 1. TROI Block Diagram

Features of TROI include:

- interactive edit, save, and load of graphical interfaces.
- real-time (20 millisecond sampling rate) data acquisition and logging.
- automated or manual communication connection to NASREM RCS.
- scripting capability to playback recorded command and parameter sequences.
- extensibility and tailorization of the data server component.
- distributed visual-interface based on X Window networking.
- embedded simulation testing procedure.

The purpose of this user guide is to provide a tutorial explaining the TROI programming environment. TROI encompasses a broad set of complex programming concepts, including: X-Window programming [9],[16]; Ada programming [2],[4]; and the NASREM architecture [1],[5]. One can use TROI without a complete knowledge of these subjects, but, it is advisable to have a basic working knowledge of these concepts. The manual contains the following sections. Section 2 provides an overview of the TROI architecture including programming constructs. Section 3 de-

scribes the programming environment including: installation of the basic TROI template, compilation and execution of a sample operator interface, and configuration of a visual interface by creating and positioning objects. Section 4 discusses the step-by-step process for customizing a TROI operator interface and covers some of the advanced features of TROI. When done with this manual, a reader should be able to build a TROI system that will communicate with the target RCS system and customize the system for any special requirements.

2 TROI Design Architecture

TROI features a multi-layered software architecture. Figure 2 details the TROI architecture and outlines the functional layering from the user to the control system. At the user level, TROI Visualization Operator interface (***TROI/VOI***) supports visualization through a workstation window-system. The TROI/VOI component is responsible for providing visual services. The Data Description Dictionary (***DDD***) contains data format descriptors. The TROI Data Server (***TROI/ DS***) connects the user-interface to RCS. Reader/writer buffers provides communication between the data server and the control system. In the control system, the control system programmer must define the command/status/ parameter reader/writer buffers and run-time connection for shared control.

2.0.1 TROI/VOI Overview

The TROI/VOI component handles the visual interpretation of data, the interactive modification to the display layout, connectivity of data to visual representations, and run-time input/output transaction processing. TROI/VOI is based on pairing producers and consumers of data. For example, a slider visual object can be used as either an output descriptor or an input mechanism. An interface design decision establishes the slider connection as either a producer of data to the control system, or a consumer of data from the control system. TROI/VOI uses the visual switchboard to define the current connectivity of the system by assigning either a producer and consumer status to either data or visual objects. Section 2.2 covers the set of graphical objects. Section 3.4 covers the interactive capabilities for manipulating graphical objects, and connecting objects to DDD entries.

2.0.2 TROI DDD Overview

DDD is the communication link between TROI/VOI and the TROI/DS. DDD is a list of the data that can be routed through the system. At run-time, DDD contains the latest transaction updates to be processed. DDD is the bridge between the control system and the user-interface. Section 2.1 covers the type of DDD descriptors available and the capabilities each provides. Section 4.1 describes the process of building a DDD.

2.0.3 TROI/DS Overview

The TROI/DS handles the routing of data to and from RCS through the NASREM reader-writer communication mechanism. TROI/DS has two purposes: move data out of RCS (sample); and move data into RCS (update). TROI/DS/SAMPLING is responsible for cyclically sampling RCS status buffers, and upon arrival of new user-interface directives, write RCS command buffers.TROI/DS/UPDATING is responsible for updating RCS rw buffers upon either external input from the user or under scripted sequencing. TROI/DS uses the data routing table to resolve physical addressing specifics of the control system rw communication interface. TROI/DS supports file operation as a part of the data routing function. File operation allows data logging and retrieval independently of the visual interface. Section 2.3 covers the NASREM reader-writer communication aspect of TROI/DS. Customization of TROI/DS is covered as a part of the manual TROI operation in Section 2.1 describing the DDD callback feature.

2.0.4 TROI Run-Time Architecture

Different needs and capabilities resulted in the implementation of TROI with a number of programming languages. The control system uses an ADA development environment. TROI uses a mixture of ADA, C and Assembly. The TROI/X module was written in C because the X environment is tailored for C. The TROI/DS was written in ADA since RCS and TROI/DS share data types. DDD is a list structure that has an isomorphic ADA and C definition.

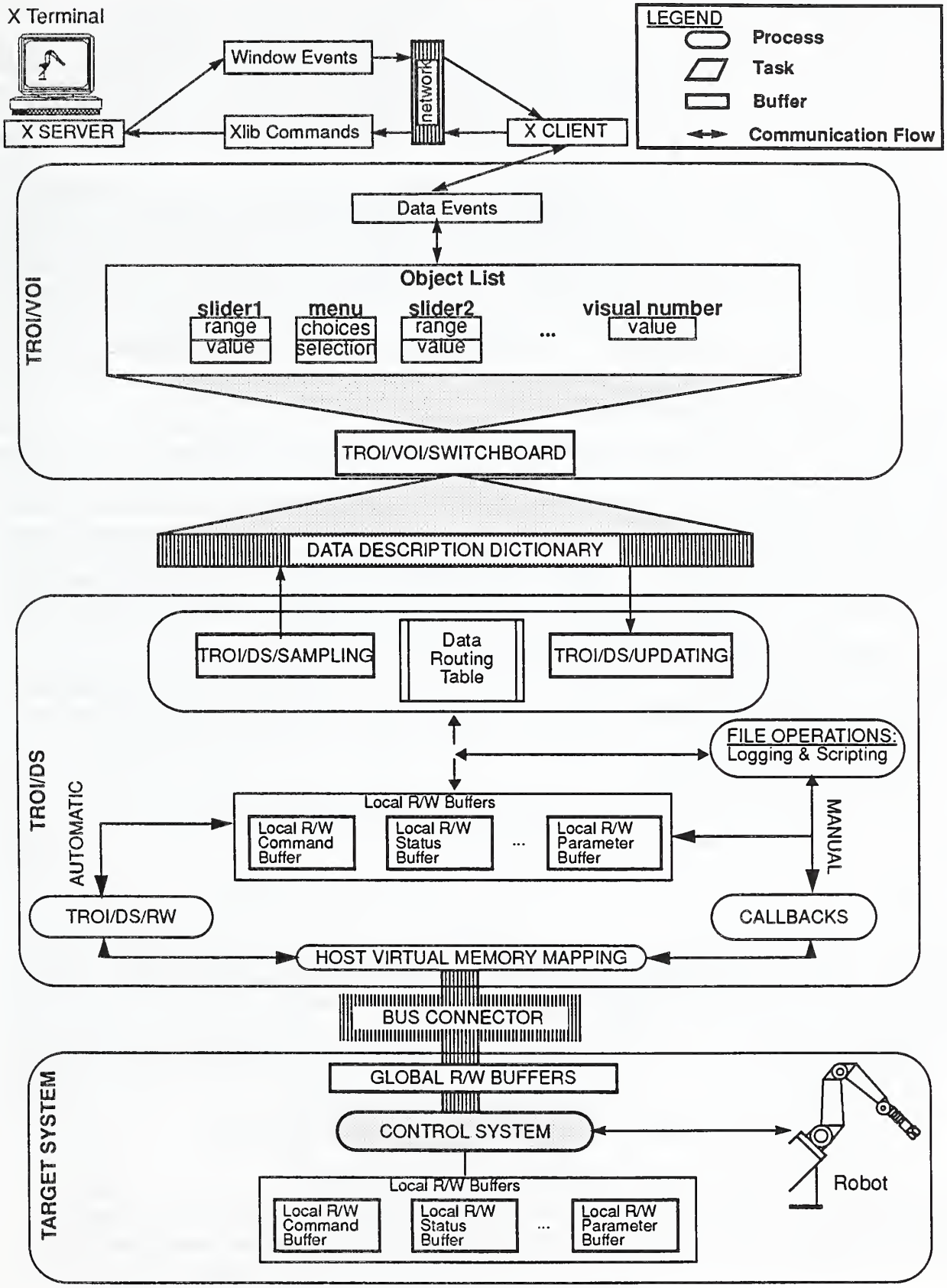

Figure 2. TROI Detailed Architecture

At run-time, TROI consists of three concurrent Ada tasks including TROI/DS/SAMPLING, TROI/VOI, and TROI/DS/UPDATING. Each of the tasks runs asynchronously and communicate through signals. An optional ***simulator task*** is provided for debugging under a completely host environment. Simulated mode transparently substitutes reader-writer communication to host-resident dummy buffers instead of addressing the target system. The simulator task reads and writes from these dummy communication buffers that would normally be sent to the target RCS system. The simulator is useful in testing a TROI system and saves much time and effort when the final target interfacing is performed.

TROI supports both ***automatic*** and ***manual*** target system communication modes. To achieve an automatic TROI system, one simply defines the DDD and then starts TROI. Each data update is automatically transmitted between subsystems. Many systems need only a one-to-one variable update correspondence. However, there are times when an entire series of variables must be set before transmitting the set of variables through the communication buffer. For example, a command buffer to a robot may require a command, a set of joint or Cartesian values, and a traversal time. Sending each of these variables upon update is questionable. Instead, the user sets up all command values, and then upon signal, the command is actually transmitted via a manual write of the r/w buffer to the target system. This form of TROI interaction is termed manual mode.

2.0.5 TROI Architecture Customization

In TROI, the control application programmer is responsible for selecting the control system variables for data routing. All data routed through the system is placed in the dictionary. Description entries in the dictionary define the format of the data, physical location in the control system, and any file connections for data logging and scripting. Using the TROI user-interface, the application programmer (or graphics design expert) independently decides the visual format of dictionary data entries. An interactive editing session defines the visual interface beforehand. At run-time, individual data is entered or displayed according the visual format definition. TROI user-interface also supports changes to the visual representation at run-time. For example, while attempting to troubleshoot a problem an operator may decide that a histogram plot conveys more information than an instantaneous bargraph.

TROI can be adapted to a variety of applications. In order to get a feel for developing aTROI operator interface, the following sections will cover some TROI management operations available. Discussed will be the TROI construction techniques, including object definitions, switchboard connection and configuration control, plus the TROI run-time data logging facility. Customizing a TROI system requires understanding several TROI concepts including:

- the dictionary type descriptors,
- the graphical visual objects, and
- RCS to Visual System Communication, i.e., NASREM reader/writer communication.

These topics will be further explained in the following sections.

2.1 Data Dictionary Descriptors

Customizing a TROI system depends on the user defining the set of important RCS variables that must be available for the user to manipulate. To customize a TROI system, a basic understanding of the parameters necessary to define a DDD element is necessary. The DDD is created at run-time and is a list of all possible variables that the operator is interested in manipulating. The DDD variables may be of the following types:

- single, double or n-dimensional numeric values, (of type 8-bit, 16-bit, and 32-bit integer; short float; double float; and duration).
- enumerated Ada type input values (these supply menu choices and selections)
- enumerated Ada type output values (these supply output string, useful in status display)
- string type input and output
- command increment (when a r/w buffer is updated, increment given variable)

The user creates a file (or modifies the sample template file) containing DDD definitions based on these DDD types with some further quantification. DDD entries are created either by compile-time generic instantiation or by a TROI data dictionary building subroutine at run-time. Some DDD entries can only be defined with generics[1].

With either DDD definition method, the user supplies a set of parameters to satisfy a DDD definition. Many DDD parameters are assigned default values and do not require specification. The user always supplies an ASCII name for the data dictionary entry (sometimes two names if bidirectional), and the variable address `VAR_ADDR` (using the Ada attribute `'ADDRESS`). Depending on the DDD type definition, and direction of data flow (either from RCS to User or from User to RCS), the remaining parameters vary. These parameter definitions are either further type definitions refinements or associated with special features of TROI, including automated communication, data logging, and callback.

Automated communication is available within TROI, but is optional. For any DDD, automated reader-writer communication (covered in section 2.3) is enabled by specifying the boolean `RW_ENTRY` condition (set to true to enable, otherwise, default is false to disable), the `RW_NAME` (this groups variables common to one r/w allowing one r/w buffer transmission per buffer update), `RW_SIZE` (to assist the table driven reader-writer communication scheme) and `RW_LOC_BUFFER_ADDR` which supplies the starting address of the local copy of the reader-writer buffer for handling the actual movement of data.

Data logging of variables from the RCS target system is available within TROI, and can be optionally enabled within a DDD definition. Data logging can be automatically enabled upon startup of TROI and the RCS target system sampling or can be enabled from the visual interface. To enable automatic data logging upon startup of TROI, the parameters FILE_NAME (requiring an ASCII path and filename string), and ENTRY_USE (set to SAVE or BOTH to enable saving) must be specified. LOGGING_TYPE is an adjoining data logging parameter specifying output style for either mode.

Callbacks are a TROI feature provided to allow extensions and are considered part of the manual operating mode. These extensions allow a broad range of customization and are powerful in addressing application-specific requirements. At DDD definition, the user supplies a callback procedures as either a routine name (for generic calls) or a routine address (for subroutine setup). Callbacks (to the defined procedure) occur every time the variable is updated either by a value update or a data logging file update. Callbacks are useful for implementing such features as:

- filters. For example, taking a user-defined angle and converting to radians before sending to RCS.
- broadcast. When one variable is set, subsequently setting other companion variables. For example, pushing one button might set three board flags.
- synchronization. When a condition occurs, initiate some action. For example, when a status variable reads done, allow a new command to be sent.
- playback/scripting. In combination with synchronization, a variable can be used to initiate playback of a script from a file by sending the recorded sequence of commands to the target system. After initiating, the synchronization feature could send subsequent commands.

Implementation examples of these callback features are contained in a sample TROI system template. (See Appendix C.)

Update is a feature which addresses the speed of data server sampling. Users can stipulate the frequency of any variables sampling rate, by specifying a duration for this parameter.

There are several generics that help build the DDD entries. It is important to remember that the DDD entries are the physical representation of a variable, not the visual representation. At run-time, the user/operator edits the visual representations of the desired DDD entries. Some specialized DDD entry builders are provided for convenience others because comparative data is not always contiguously represented. For example, a 2-D plot would require two variables that may not be contiguous in the application data space, but are a subset of the n-dimensional (*ND*) data descriptor space. Thus a programmer selects fields from one record and pairs them with another to achieve the x & y type representation.The current set of DDD entry builders includes:

- `NUMERIC_1D_USER_TO_RCS_BUILD` - build 1 number data routing from user to rcs.
- `NUMERIC_1D_RCS_TO_USER_BUILD` - build 1 number data routing from rcs to user.
- `NUMERIC_2D_USER_TO_RCS_BUILD` - build 2D (x & y) data routing from user to rcs.
- `NUMERIC_ND_RCS_TO_USER_BUILD` - build array of numeric values data routing from rcs to user.
- `MENU_BUILD` - build enumerated type selection routing from user to rcs

1. Generics allow passing of Ada types and subroutines don't.

```
AUTOMATIC:
package OI_PIPE_LVL2_CENTROID_ARRAY is new OI_X.NUMERIC_ND_RCS_TO_USER_BUILD
    (
    NAME                => "Centroid X & Y VALUES : ",
    ARRAY_ADDR          => OPINT_PIPE_LVL2_CENTROID.CENTX'ADDRESS,
    DATA_TYPE           => OI_X.SHORT_FLOATING,
    NUM_IN_ARRAY        => 2,
    RW_ENTRY            => TRUE,
    RW_NAME             => "OPINT_PIPE_LVL2_CENTROID ARRAY",
    RW_SIZE             => OP_INT_CENTROID_TYPE'SIZE,
    RW_LOC_BUFFER_ADDR  => OPINT_PIPE_LVL2_CENTROID'ADDRESS,
    FILE_NAME           => "centroid.dat",
    ENTRY_USE           => OI_X.SHOW,
    CALLBACK            => FILECALLBACK
    );
MANUAL:
package OI_PRIM_REDUNDANCY_RES_TECHNIQUE is new OI_X.MENU_BUILD
    (
    NAME_IN             => "Redundancy Resolution Technique Selection",
    NAME_OUT            => "Redundancy Resolution Technique List",
    VAR_ADDR            => OI_PRIM_COMMAND.REDUNDANCY_RES.TCHNIQUE'ADDRESS,
    ENUM                => PRIM_REDUN_TECHNIQUE,
    );
```

Figure 3. Automatic vs. Manual r/w DDD Package Instantiation

- ENUM_RCS_TO_USER_BUILD - build enumerated string output routed from rcs to user
- CMD_INCR_BUILD - build command number increment with r/w on updates

An example of the disparity in declarative parameters for a generic instantiation is illustrated in The OI_PIPE_LV-L2_CENTROID_ARRAY is a generic n-dimensional numeric package build. (At run-time this package will be initialized causing the DDD table entry to be created.) It describes an array within a reader/writer buffer that is automatically communicates after every update. OI_PRIM_REDUNDANCY_RES_TECHNIQUE, on the other hand, is a variable from a reader/writer buffer that would depend on some other VOI variable to trigger a callback that would manually send the communication buffer to the target RCS system. Figure 3 illustrates the differing styles of parameter instantiation.

Complete parameter templates of these DDD entry builders is given in Appendix D.

There is only one TROI procedural method to install a DDD entry. This function is the ***n-vector build*** taking n-dimensional vectors as a parameters. It takes the regular TROI feature arguments, but also has the additional parameters: DATA_TYPE, DATA_FLOW_DIRECTION, FILTER_CALLBACK parameters (VECTOR_ADDR is merely a rename of VAR_ADDR). The DATA_TYPE specifies the numeric representation which can be either integer (LONG), float (SHORT_FLOATING), double (DOUBLE) or duration (TIMED)[2]. The DATA_FLOW_DIRECTION parameter specifies the connectivity of the DDD entry, whether the information flows from RCS to visual interface (RCS_TO_USER), or vice versa (USER_TO_RCS). FILTER_CALLBACK is a specialized callback feature for data filtering that uses a set of predefined TROI data filters. Figure 4 shows the calling protocol of N_VECTOR_BUILD and then a typical invocation. Note the use of defaults in the calling specification means that many of the calling parameters do not need actual values. (In MANUAL mode the defaults assumed by the specification are RW_ENTRY => FALSE, RW_NAME => "NONE", RW_SIZE => 0, RW_LOC_BUFFER_ADDRESS => address'-ref(0), FILE_NAME => "NONE", CALLBACK => NULLCALLBACK).

2.2 Visual Operator's Interface Graphical Objects

The *Visual Operator's Interface* (*VOI*) provides front end access to the list of DDD variables extracted from RCS. It is through manipulation and display of these variables that the user is able to change the internal state of the RCS, and

2. the enumerated NUMERIC_TYPE specification is in parentheses.

gather information about the state of the RCS. The VOI provides ***snapshots*** of the state of the RCS, that is, states are not queued up and thus some information may be lost between snapshots. The VOI and the RCS run concurrently, and the VOI, due to the nature of embedded X window system library, will typically not be able to display information as fast as this information is produced. However, when the TROI does take a snapshot of the system, it is guaranteed to be an accurate, up-to-date snapshot at the instant the "picture" was taken.

The VOI has its own interface to the data description dictionary (DDD). The DDD defines the list of ***system variables*** necessary for the user to manipulate the control system. The VOI and the RCS data server communicate by transferring data via the DDD which is set up by the application RCS process. The DDD variable list is shared between VOI and RCS, and though configurable on the RCS side, this list is static through the life of the VOI session. System variables are unidirectional, though some variables may have an undetermined direction when they are created, and others may change direction under some circumstances. The direction of the variable is determined when the variable is connected into a ***data flow***.

The VOI maintains its own ***dictionary*** consisting of those variables associated with system variables, and other variables attached to the various VOI objects which the user creates interactively. These VOI variables are also unidirectional, and again, the direction of some variables may be undetermined when they are created. The terms ***consumer*** and ***producer*** are used to describe variables whose data flow direction have been determined. A consumer is a variable into which data flows; that is it consumes information. Similarly, a producer variable generates data which flows out of the variable. By connecting a producer to a consumer, the user configures the data flow of the VOI system. A single producer may be connected to any number of consumers, and each consumer will see an exact replica of the information at each clock tick. A single consumer may only consume information from one producer. The structure which maintains the data flow connections in the VOI is called the ***switchboard***.

The user configures the desired operator interface by creating, modifying and destroying ***objects***. Some objects have visible components (e.g. sliders, or menus) and these are called ***visual objects*** (abbreviated as *vo*'s, pronounced "*Vee*

```
Routine Specification:
procedure N_VECTOR_BUILD
    (
    NAME                : in string;                          -- name of the data routing entry
                                                              -- Reader/writer specs
    VECTOR_ADDR         : in system.address;                  -- address of array to be routed
    DATA_TYPE           : NUMERIC_TYPES := LONG;              -- numeric representation
    NUM_IN_VECTOR       : integer:=1;                         -- number or elems in vec array
    DATA_FLOW_DIRECTION: CONNECTION_TYPES := USER_TO_RCS;     -- which way does the data go
    UPDATE              : INTEGER:=0;                         -- update rate
                                                              -- Numeric attributes
                                                              -- Reader/writer linkages
    RW_ENTRY            : boolean :=FALSE;                    -- reader/writer entry? assume not
    RW_NAME             : string := "NO_RW";                  -- name of the reader/writer
    RW_SIZE             : integer := 0;                       -- size of read/write buf in bits
    RW_LOC_BUFFER_ADDR : in integer :=0;                      -- local read/writ buf address

                                                              -- Data routing parameters
    FILE_NAME           : string := "NONE";                   -- default datalog filename to
                                                              -- save values
    PROC_CALLBACK       : ADDRESS:= CALLBACK'ADDRESS;         --
    FILTER_CALLBACK     : ADDRESS:= FILTERCALLBACK'ADDRESS; --
    LOGGING_TYPE        : LOGGING_TYPES := TEXT

Invoking with actual parameters:
OI_X.N_VECTOR_BUILD(
    NAME                => "Prim Joint 1",
    VECTOR_ADDR         => LOCAL_TERMINATION_COND.JOINT_GOAL_POSE(1)'ADDRESS,
    DATA_TYPE           => OI_X.SHORT_FLOATING
    );
```

Figure 4. *N_VECTOR_BUILD* ***TROI DDD Procedure***

Oh's"). Other objects do not have any visual components. These non-visual objects are often ***filters*** which manipulate data along the data flow path. All objects are given names by the user when created, which must be unique. All objects also have some variables attached to them through which the object consumes or produces information. Each object defines its interface to the system through its attached variables and it is by connecting these variables that the user achieves input or output between the interface and the RCS.

Every variable in the VOI dictionary has a ***fully scoped*** name which uniquely identifies it in the system. A fully scoped variable name contains both the variable's local, possibly non-unique name and the name of the object to which the variable is attached. The fact that object names are unique guarantees that fully scoped variable names are unique. Variable names are usually predefined by the objects which contain the variable. For example, sliders have a *value* variable which contains the value produced by the slider. All sliders will name their value variable "*value*", and it is the slider object's user supplied instance name which makes the variable unique. Fully scoped variable names are displayed in the VOI with this format: *<ObjectName>:<VariableName>*, where the colon (":") is part of the fully scoped name.

The VOI is both data driven and event driven. As data changes, the VOI recognizes this and propagates these changes along the data flow path. As the user interacts with the various vo's in the interface configuration, these window system events are recognized and acted upon. The VOI must merge these two separate event paths and does so by interlacing the window event loop with the data event loop. The structure which performs this interlacing is called the ***watcher*** and the interlace interval is user configurable.

Below is a list of currently implemented objects, along with a description of the object's features. Examples of these objects can be seen in Figure 5.

2.2.1 Visual Interface Object: Alarm Box

The ***alarm box*** object is used to display a string to the user, but the most interesting feature of this object is that the string resides in a separate frame, which can be dismissed and/or positioned by the user as a separate X window. Because this vo actually resides in a separate frame, it will not appear in the Operator's panel is thus not subject to the same rules of overlapping as are other vo's. This object contains a single string variable, called *string* which only consumes data.

When the string value is modified, the alarm box will pop up in its last known position on the screen and display the string in the message area. The alarm box contains a *Dismiss* button which pops down the frame until the string value is once again modified[3].

Figure 5 includes two examples of alarm boxes. They are labeled *2-feedback* and *4-feedback*. You can see how they are actually separate X windows from the Operator's Panel.

The VOI system creates a special instance of the alarm box object which is used to inform the user of operating errors as they occur (for example, connecting two variables of different types). The user can set this object's resources like any other alarm box object. Its instance name is ***[SystemAlarm]***.

2.2.2 Visual Interface Object: Cursor Tracker

The ***cursor tracker*** object allows the user to enter 2D positional information by using the cursor. Within the tracker frame, two cursor tracking methods can be employed. Under the first method (called *Button Down Only* mode), X and Y positional information is only produced when the left or middle button is held down and dragged within the frame. Under the second method (called *All Cursor Motion* mode), any cursor position (within the cursor tracking frame), regardless of button state, generates X and Y positions. The tracking method can be changed by interacting with the menu which pops up when the user clicks the right button within the cursor tracking frame. In Figure 5, the window labeled *3-tracker* is an example of a cursor tracker object. Like alarm boxes, cursor trackers are vo's which don't appear in the Operator's Panel.

The coordinate system inside the frame places the origin at the upper left corner, with positive X growing to the right and positive Y growing down. This corresponds to the X window coordinate system.

3. These behaviors will depend on the X window manager you are using.

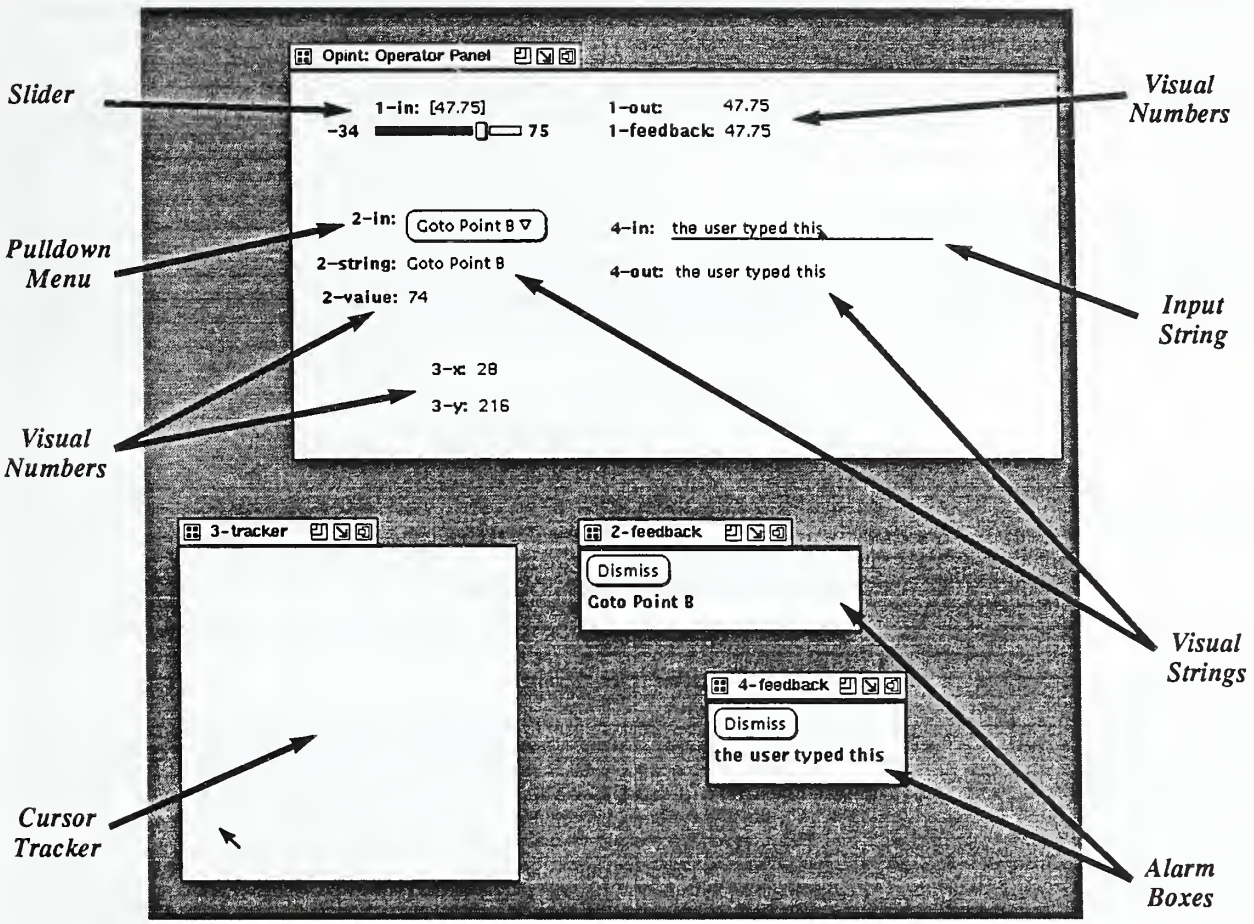

Figure 5. Sample Operator Screen with Visual Objects

The cursor tracker object contains a single producer variable called *position*, which is a 2D NVector type variable. The X value is contained in slot zero and the Y value is contained in slot one.

2.2.3 Visual Interface Object: Input String

Input String objects are used to enter text strings from the user. These are very similar to other text input fields found in other parts of the VOI control frames. The standard editing control keys apply within the field, with two additions: the *control-C* key and the *return* key.

The return key ***enters*** the string's value into the system, and until it is typed, the string is not propagated from the input string object's producer variable. The control-C key is a ***cancel*** key which resets the text field to the last entered value.

The input string object contains a single producer variable, called *value*, of type String. The input string's visual components consist of a label and a text field. An example of an input string object can be found in the operators panel in Figure 5. The object labeled *4-in* is an input string object.

2.2.4 Visual Interface Object: Pulldown Menu

The ***pulldown menu*** object allows the user to make a selection from a list of choices. The visual object associated with these choices consists of a label and a pull down menu button. The menu button displays the last chosen selec-

tion within the button image. Clicking on the button with the left mouse button indicates the user has re-selected the last selection. This is a convenience so the user does not always have to pop up the menu to make multiple re-selections. Clicking on the right button, brings up a menu displaying all the choices available to the user. The previous selection is highlighted with a ring around the selection text. An example, labeled *2-in* can be seen in Figure 5.

The pulldown menu contains two variables, a consumer called *choices* from which the menu reads the list of available choices. Naturally, it is of type Choice. Also, the menu contains a producer variable of type Selection, called *selection.*

2.2.5 Visual Interface Object: Selection to String/Value Filter

A variable of type Selection is a compound variable containing both a string and an integer value. There are no objects which will directly display this type of variable, so the ***selection to string/value filter*** (also called a ***selstr filter***) is employed to convert a Selection type variable into two separate data streams (See Figure 6). The selstr filter has no

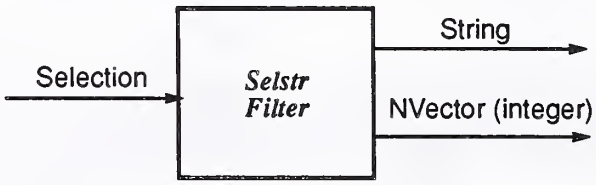

Figure 6. Selstr Data Flow Split Filter

visual components.

This filter contains three variables, a consumer of type Selection, called *selection*, and two producers, one of type String, called *string*, and the other called *value*, which is a one dimensional NVector. The integer value is placed in slot zero of the NVector variable.

In Figure 6, a selstr filter is employed in the data flow from *2-in* to *2-string* and *2-value*. The selection produced by *2-in* flows through a selstr filter called *2-filter* and is split into a string variable and integer value variable. The string variable is connected to the consumer of *2-string* (a visual string object) and the integer variable is connected to the consumer of *2-value* (a visual number object).

2.2.6 Visual Interface Object: Slider

The ***slider*** object allows the user to enter numerical values from a bounded continuous range. Sliders can also be used to display numerical values back to the user, though they are not the best object for user feedback. Sliders can manipulate double precision floating point or integer values.

The slider consists of five visual components: the label, the value area, the maximum and minimum range values, and the slider bar area. The value area simply shows the exact numerical value of the slider. The slider's value can be changed by grabbing the handle within the slider bar area with the left mouse button and dragging the handle back and forth. The value area will change as the slider handle is dragged, but the slider's producer variable will not be updated until the slider handle is released. An example of a slider is the object labeled *1-in* in .

The slider contains two variables: a consumer called *range* and a variable called *value* which has an undetermined direction when created (i.e., it can be connected to either a producer or a consumer). The range variable is a 2D NVector variable with elements of type double. It is from this variable that the maximum and minimum range values can be supplied by the RCS process (if this variable is not connected to a producer, a default maximum and minimum will be supplied), or can be supplied as a resource (see section 4.3 on TROI resource management). The minimum value resides in slot zero of the NVector and the maximum resides in slot one.

The data flow direction of the *value* variable is undetermined at the time of creation, which allows the user to connect it to either a consumer or a producer. Once the connection is made, the direction of data flow is set and cannot be changed until the connection is broken. The *range* variable is a 2D NVector.

2.2.7 Visual Interface Object: System Variable

The *system* object is a wrapper object that contains a single RCS System Variable as described as a part of the DDD. This object converts and transports data from VOI internal structure through to the RCS structure. It also handles reading new data from the system and transporting it into VOI through the switchboard mechanism.

The operator cannot create, modify, or destroy system objects. Creation is accomplished automatically when the system starts up, through the RCS's initialization procedures. System objects are persistent through the life of the VOI session. The only time you will come in contact with system objects is when you connect to or from them using the Connections frame, or when you perform data logging on them.

When a system object is created from a system variable in the RCS dictionary, VOI uses a modified version of the RCS dictionary name as the name of the object. The variable name is modified by surrounding it in angle bracket characters ("<" and ">"). If this does not yield a unique object name, then a unique integer identifier is appended to the variable name before the brackets are attached. The brackets also serve to visually identify system variables in the consumer, producer, and data log lists.

2.2.8 Visual Interface Object: Visual Number

The ***visual number*** object is used to display a numeric value to the operator. The value can be in any one of these three forms: integer, float, or double (as defined in C). The visual number object contains two visual components, the label and the value field, which are used for screen output only. No input from the user is recognized by this object. Figure 5 contains many examples of visual number objects: *1-out*, *1-feedback*, *2-value*, *3-x*, and *3-y*.

The visual number object contains a single consumer variable, called *value*, of type 1D NVector, with the value in slot zero.

2.2.9 Visual Interface Object: Visual String

The ***visual string*** object is very similar to the visual number object, though instead of being used to display a numeric value to the operator, it displays a string value to the operator. Like the visual number object, this object is used for output only. Figure 5 also contains a few examples of visual string objects: *2-value* and *4-out*.

The visual string object contains a single consumer variable called *value*, of type String.

2.2.10 Visual Interface Object: XY Position Filter

The cursor tracking object as described above, produces a combined, 2D NVector variable describing the current position in X and Y coordinates. No output object can directly display a numeric vector variable of depth greater than 1, so the ***XY position filter*** object is employed to split the single data stream into two distinct data streams, one for the X value and the other for the Y value. The function of this filter is very similar to the selstr filter described above.

This filter has no visual components. It does contain three variables: a consumer variable called *XY in*, which is a 2D NVector of the type produced by a Cursor Tracker object; a producer called *X out*, which is the X value stream in a 1D NVector variable; and a producer called *Y out*, the Y value stream, also a 1D NVector.

This filter, having no visual components, does not have a resource class, nor does it recognize any resource settings.

2.3 Reader-Writer Buffer Communication

A communication mechanism is embedded within TROI that enables a visual interface to share data with a NASREM target process. NASREM communication follows a reader-writer methodology [3],[7]. TROI communication is done via the NASREM reader-writer communication scheme [5]. In the automated mode, any data updates or modifications by either side, results in the opposite process being informed of the update. TROI features target communication via the NASREM reader-writer communication scheme, but TROI is not limited to this communication scheme. The user may extend the system via the callback mechanism to allow other communication strategies. However, for typical users, the NASREM reader/writer communication scheme forms the basis of user-to-target-system communication. Understanding NASREM reader/writer communication is important in building a TROI system.

The NASREM reader-writer package uses global data buffers for communication and local buffer images for manipulation. The local buffer images are written/read from the global data buffer during communication. The NASREM reader-writer package exploits Ada generic instantiation to achieve a simple commonality among communication buffers[4]. The Ada generic construct offers a flexible way for providing a set of operators (READ, WRITE, INIT, DATA_READY, etc.) that can be applied to diverse set of buffer types. With this generic capability, NASREM communication is achieved by having two processes use the same type definition, declare generic instantiation supplying identical physical addressing, and then use the generic tool operators to communicate through the buffer. These generic declarations facilitate intra and inter-processor communication.

The NASREM reader-writer communication scheme is best illustrated by example. Initially, a list of buffer names defines the user reader-writer communication pool. (This pool of names differs from user to user, although obviously there is an overlap of logical to physical addressing to accommodate the sharing of data.) This naming convention offers a convenient mechanism for assigning physical addresses to logical reader-writer data abstractions. In the basic sample TROI system the list of buffer names are:

```
type buffer_names is (
                      PAGE_BOUNDARY,
                      RW_COMMAND_BUFFER
                      RW_STAUTS_BUFFER,
                      RW_PARAMETER_BUFFER
);(where PAGE_BOUNDARY is a unique name to help align the virtual memory mapping.)
```

Then communication buffer types are defined. These buffer types may be built as composite element types. All variations of Ada type definitions are permissible within a NASREM reader-writer communication buffer. For example, in the following, PROCESSING_MODES is an enumerated dependent type specifying the command modes, while COMMAND_TYPE specifies the actual command layout. COMMAND_TYPE is what will be use to form the generic instantiation of the command reader-writer buffer.

```
type PROCESSING_MODES is (SALUTING, AT_EASE, MARCHING);

type COMMAND_TYPE is
     record
                      COMMAND_NO                    : INTEGER;
                      MODE                          : PROCESSING_MODES;
                      SPEED                         : DURATION;
                      ENERGY                        : SHORT_FLOAT;
   end record;
```

With a generic ***reader-writer*** (***r/w***) package instantiation, the r/w communication buffer declaration is then defined. It uses the command new READER_WRITER to create the new generic, and uses the DATA (for the buffer type) and NAME (to resolve the logical to physical addressing) elements as descriptors. The following demonstrates a simple generic package instantiation example.

```
package RW_COMMAND is new READER_WRITER (DATA => COMMAND_TYPE
    NAME => RW_COMMAND_TYPE);
```

The user also must also declare a local buffer for manipulation:

```
COMMAND: COMMAND_TYPE;
```

With the local instantiation, the user first assigns values to the local buffer copy:

4. We assume that the reader is familiar with the Ada generic construct. See[2][4] for more information.

```
LOCAL_COMMAND:=
    (COMMAND_NO          => 0,
     MODE                => AT_EASE,
     SPEED               => 0.2,
     ENERGY              => 10.0
    );
```

and then initializes the global buffer with the generic Ada operator syntax..

```
RW_COMMAND.INITIALIZEE.(LOCAL_COMMAND);
```

Now, the NASREM reader-writer communication scheme is enabled and the user is free to read, write, and query the global data buffer via the generic operators `READ`, `WRITE`, and `NEW_DATA`. For example, to send a new command, all the parameters could be maintained (or changed) but the command number would be increased.

```
LOCAL_COMMAND.COMMAND_NO := LOCAL_COMAND.COMAND_NO + 1;
RW_COMMAND.WRITE.(LOCAL_COMMAND );
```

TROI supports this mechanism in two modes, either automatic or manual mode. In the automatic mode, TROI supports bi-directional data traffic:

- the data server reads new data from the RCS target system via a r/w buffer that is updated in the corresponding r/w data elements in the data dictionary, which the VOI acknowledges. This direction of information transmittal is called TROI/DS/SAMPLING.
- the TROI visual interface accepts user input which updates the corresponding data element in the data dictionary that the data server automatically writes to the local r/w buffer. In automatic mode, the data server will do a write-through to the target-system RCS r/w buffer. In automatic mode, no specialized code need be supplied. Rather, upon DDD entry declaration, a set of r/w parameters are supplied. TROI also supports manual reader-writer communication that is triggered only upon some user or target event that is under the direction of non-system TROI code (supplied within a callback feature.)

In order to allow automatic r/w communication and save a complete recompilation of the TROI support Ada library, TROI reader/writer (TROI/RW) is a modified version of the NASREM reader-writer mechanism. (Specifying a new set of buffer names has a large set of dependent files that would need recompilation - which would imply recompiling all of the TROI system code). The NASREM scheme (TROI manual mode) distributes reader-writer service throughout the system. In automatic mode, TROI centralizes reader-writer service with a table-driven reader-writer mechanism. After declaration, the TROI/RW is completely compatible with the regular NASREM r/w mechanism (and actually shares the same code.) TROI uses the same enumerated type buffer definitions:

```
type buffer_names is (
    PAGE_BOUNDARY,
    RW_COMMAND_BUFFER,
    RW_STATUS_BUFFER,
    RW_PARAMETER_BUFFER
    );
```

However, all DDD definitions files has code to allow a one-to-one mapping of enumerated type into a positional element in the array `buffer_addr`. In the original NASREM `buffer_addr` is an array of the enumerated buffer names. TROI declares `buffer_addr` as an array of integers, and then uses the enumerated elements as positions in this array. To use this enumerated positioning scheme in an integer array the following Ada unchecked conversion is necessary:

The enumerated positioning scheme in the `buffer_addr` integer array is automatically used by the TROI logical to physical address mapping code. However, because TROI can either be a target-buffered system or be a self-contained simulated buffer scheme, each r/w buffer must be reset after program startup and ADA package instantiation to then decode the command line options to physically position reader writer buffers. This is achieved by using the TROI ex-

```
for buffer_names'size use 32;

function to_int
    is new UNCHECKED_CONVERSION
    (SOURCE => buffer_names, TARGET => integer);
```

tension r/w operator RESET. This is the only time the enumerated to integer conversion must be undertaken and the following shows an example:

```
RW_COMMAND.RESET( BUFFER_ADDRESSES.buffer_addr(to_int(RW_COMMAND_BUFFER)));
```

Any r/w operation within a TROI callback extension can then perform any normal reader-writer operation, such as:.

```
RW_COMMAND.INITIALIZEE.(LOCAL_COMMAND);
```

In target mode transparent to the user, the host and target system are connected by a host virtual memory mapping scheme resolved internally in the TROI system. TROI added the enumerated `buffer_type` `PAGE_BOUNDARY` to assist in defining the host to target memory mapping. Upon TROI startup, a user file defining r/w logical naming to physical addressing file is input. In the NASREM r/w scenario each enumerated name is paired with a physical address. In the TROI r/w scenario, a base address and memory size is paired with `PAGE_BOUNDARY` to define a section of target memory containing r/w buffers, then each enumerated name is paired with an offset into this memory space. The base address and offset addressing scheme supplies compatibility with the UNIX `mmap` command.

3 TROI Programming Environment

Using TROI is relatively simple. However, TROI is built using the X window methodology. Therefore, one must have a general understanding of the X programming environment. The first section gives a brief X background and defines general X terminology. The TROI system is built (compiled and linked) using an Ada software environment, so that it is necessary to understand the basics of the Ada terminology and programming environment. Given this background, the following sections will discuss building and using a sample TROI system. These sections include descriptions of the interactive capabilities of TROI including editing and modifying the visual interface, and sending and receiving data from the control system.

3.1 X Programming Environment

This section will provide a very brief background of the philosophy of X. It is important to understand the levels of abstraction in an operator interface. Current graphics technology ranges from computers supporting graphic engines to personal computer windowing-systems. The disparity of the graphics technology base and the need for a standardized interface results in the need to select a flexible, portable, and widely-supported Graphical User Interface Development Environment (*GUIDE*).

A GUIDE is a hierarchy of graphical abstractions [10]. The modularity of a GUIDE provides portability across a wide range of machine architectures with modifications restricted to one level in the GUIDE hierarchy. The levels of the GUIDE hierarchy from the highest to lowest level of abstraction are:

- the user
- high-level development shells
- user interface management systems (*UIMS*)
- graphical user-interface
- style guides
- toolkits
- windowing system

- operating system
- hardware

TROI uses the 32-bit engineering workstation as the computer platform. Most 32-bit engineering workstations use the UNIX[5] operating system. UNIX provides networking capabilities which implies the need for a networked windowing system. The X window system[6] developed at MIT is widely-supported, public domain software that provides graphical networking. Although X does not feature high-resolution graphic support, X has such broad industry support that it has become a de facto standard. X alone does not define the graphical interface. Within the X world, numerous toolkits are available to augment the functionality of X. TROI uses the ***XView*** toolkit which presents an OPEN LOOK style window GUI environment. XView was selected because of previous experience with SunView[7], however, because of the insularity of GUIDE hierarchy, some future transition to alternative window GUI environments (e.g. MOTIF[8]) is possible.

3.1.1 X Window System

The X Window System is based on a client/server model. The X server supplies the primitive graphics routines while clients send packets of instructions to the X server for graphical display. The client/server model of X provides distributed graphical communication so that client graphic programs can run across machine boundaries.

The basic data object in X is the widget which encapsulates graphical information with an object-oriented approach. Sets of widgets are defined as toolkits. To the application programmer, a toolkit is a library of graphical object subroutines. A sample of these objects include menus, slide bars, charts, buttons and alert fields. A style guide is responsible for defining the toolkit's visual representation and interactive behaviors. For example, the menu widget displaying a pop-up style as opposed to a pull-down style. Figure 7 illustrates some of the basic X windowing terminology.

Although helpful, managing the visual objects within an X toolkit is time-consuming and specialized. Minor changes in visual presentation do not map into minor programming changes. For this reason, the separation of application data from graphic visualization is imperative in a robust and flexible operator interface. Within the GUIDE hierarchy, it is the UIMS level that is responsible for handling the requirement for the separation of application and user functionality. TROI adheres to the GUIDE hierarchy in design and can be considered a UIMS.

3.2 Ada Programming Environment

This section will provide a very brief background on the Ada programming environment in support of TROI. Ada is a general-purpose programming language developed under the initiative of the United States Department of Defense. Its goal was to satisfy the programming needs of large, real-time embedded systems. It is important to understand some of the basic terminology of Ada. Ada terms ***data types*** as a means for describing the structure of data. ***Declarations*** create actual instances of a type. For example, whereas `float` is a type, `X : float` is an actual declaration. Ada supports ***packaging*** (or ***encapsulation***) of related programming constructs to give an object based programming flavor to the language. (It is not truly object-oriented since the language does not directly support either classes or inheritance.) Packaging includes both a ***specification*** detailing the semantics of the package (such as the fact that +, -, /, * are available operators to `floats`) and a ***body*** containing the implementation (which contains either a software or hardware definition hidden from the user). Packaging rationale is that a specification is readily available to the user, but information-hiding precludes the need to know the details of the package body. Ada supports ***tasking*** as a part of the language in order to support concurrent programming. Ada offers a ***generic*** capability to define general templates that can then be instantiated to describe a variety of actual packages or subprograms.

TROI runs under the VERDIX Ada environment or ***VADS***[9]. VADS is a DOD validated version of Ada. VAD processes the full Ada language as specified by the Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A[4]. VADS supports both self-hosted Sun-3 system and target-based 680x0 systems. TROI runs under a Sun-3

5. AT&T trademark.
6. Massachusetts Institute of Technology Consortium trademark.
7. Sun Microsystems, Inc. trademark.
8. Open Software Foundation trademark.
9. Verdix Corporation trademark.

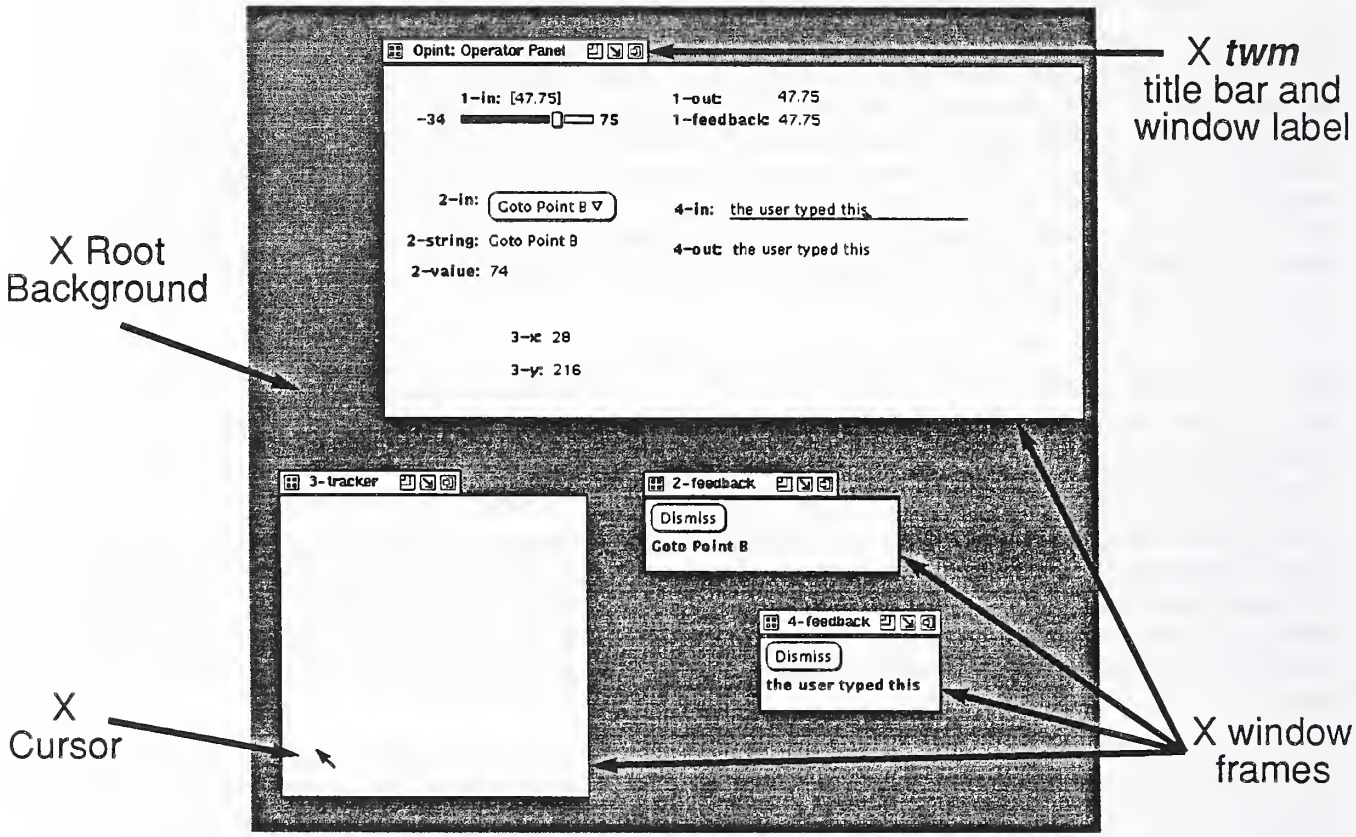

Figure 7. X Window Environment Terminology

host-based Ada environment and communicates to 680x0 target-based Ada environments in real-time across a bus-connector.

Any TROI program must run on a machine linked to the target hardware with a bus connector. TROI contains code which maps a physical address of the bus connector into a virtual host address to support this connection. TROI is running a three process input, process, output tasking model with an optional simulator task. TROI uses Ada wrappers to interface to the C world of X.

VADS requires all compilations to take place in an Ada library. Any directory may become such a library, called a VADS library in the VADS user guide. In a working VADS directory, the file *ada.lib* contains many of the directives important in understanding the compilation and linking strategy. The *ada.lib* file the `ADAPATH` line contains the search path the compiler uses for package lookup. When you `WITH` a package, you must insure that the location of the package is contained on the `ADAPATH` line, and that the package specification (and body) have been compiled, and available for search in their Ada library directory:

```
ADAPATH= /home/stella/ada/self5.7/verdixlib /home/stella/ada/self5.7/standard
ADAPATH= /home/stella/michalos/ada/src/opint/current ~/opint/sample/code
```

The link options are set using `WITHx` directives. Several of the `WITH` commands are already set by VADS so that the next new `WITHx` must follow sequentially after the largest `WITHx` from all the `ADAPATH` directory *ada.lib*'s that are actually used. This is an example of the configuration of the `WITHx` necessary to run a sample TROI system.

In this example scenario, the sample ada.lib contains `WITHx` commands that skip from 1 to 4 because system TROI packages use `WITH2` and `WITH3`.

```
WITH1:LINK:/home/stella/michalos/ada/src/opint/usr_config/.objects/v_usr_conf_b01:
WITH4:LINK:/home/stella/michalos/ada/src/opint/x/opint:
WITH5:LINK:/home/stella/michalos/ada/src/opint/current/oi_C_utils
```

3.3 Building the new TROI Ada enabled Directory: *a.troi_build*

Initially, the UNIX C-shell script *a.troi_build* is used to automate the basic directory setup of a new TROI system. The user must create a new directory and then change to this directory. The user then invokes the command *a.troi_build* which performs several basic tasks. First, an Ada library supported directory is created:

```
mkdir myTROI
cd myTROI
```

Second, a set of sample TROI files are copied into the new oi directory via the `TROI` command `a.troi_build`. These sample TROI files include sample data definitions containing reader/writer definitions, callback functions, a simulator to test the sample TROI, and a main routine `oi.a` that defines the sample dictionary, and also contains run-time code. This completes the basic directory setup.

```
a.troi_build
```

It is the responsibility of the user to copy application-specific files containing reader-writer communication buffers declarations into the new directory . To simplify construction and streamline compilation of a TROI system, the user can reference the type and reader-writer packages in another directory. Then the new TROI *ada.lib* can be modified to reflect the ability for lookup of packages in these directory. One can simply reference existing packages, but must insure that the compilation of these packages has been performed *with a self compiler!* To modify the new TROI directory to access these packages, the Ada command `a.path` is used to insert the package directory's path into the new Ada library path lookup.

```
a.path -i "package_directory_path"
```

Otherwise one can create a Ada subdirectory off the new TROI directory, copy in the applicable code, compile this code and simply reference the packages contained in the new subdirectory. The option code in the *a.troi_build* will automatically build a new Ada library based subdirectory called *code* off the new TROI directory. The following command performs the new subdirectory creation automatically:

```
a.troi_build code
```

Then the user can copy in the relevant files into the directory *code* and compile these fields under the self hosted compiler. It is advisable to modify the command script file *doit* to reflect the files copied, so that later, should the corresponding target-based files change, these files can be automatically copied over to the subdirectory *code* and then recompiled.

Within the file *doit* in the new TROI directory, the copy option can be modified to reflect the dependency relationship between files:

```
#
# Copy over all the necessary real data definitions
#
copy: cp /...some_path.../*.a ./code
```

Later, copy updates can be performed with by invoking *doit* with the *copy* option:

```
doit copy
```

3.4 Sample TROI Session

This section will illustrate the basic mechanics of compiling, linking and running the sample TROI system. To achieve a successful `oi.sample` program a sequence of steps is required. First the Ada programs must be compiled on a host with licence to run the Ada compiler. The compilation phase is done with the command *doit oi:*

```
doit oi
```

Problems that may occur are the lack of virtual memory on the host machine. If this occurs, remove some applications from your window environment. Similarly, check the */tmp* directory for stray files and for any hidden files (starting with a leading period ".")

Then the files must linked with the necessary Ada and X libraries. To do this run the *doit* command with the *link* option. This command can be run on any machine, but currently requires a machine running SunOS 4.0 or later for successful operation:

```
doit link
```

To test the sample TROI system, use the command *runit*. This will bring up the basic run-time system that can be used for exploring the various visual interface features:

```
runit
```

Although not apparent from this sample session, TROI supports two modes of operation, ***self*** versus ***target mode*** TROI. In the target mode, TROI allows a direct connection to the target system for communication. However, the target mode is a poor (and possibly dangerous - especially if connected to a robot) debugging environment. For this reason, when debugging it is more expedient to flesh out a simulator (which in its simplest form just echoes the input target system command, and then emulates some status feedback.) Although a simulator merely checks that eventual host-target r/w communication is valid, at integration time this testing save much time and effort.

This system is demonstrating the self mode of the TROI system. The system is not connected to an actual target system, rather it uses a simulator for feedback. The basic sample system shows two visual objects that should always be included in your visual interface switchboard: the *TROISystemState* and the *TROISystemSpeed*. These control the sampling of target system inputs. The TROI sampling system is initially disabled with the sampling speed set to access every 100 milliseconds. These can be changed to reflect different run-time configuration. When toggling between the *IDLE* and *RUNNING TROISsytemStates*, the TROI console issues an "Input system asleep", or an "Input system awake" message. (Note, that if the *TROISystemSpeed* is set to a large amount, it will take a longer time before the sampling system responds to *IDLE* and *RUNNING* commands and issue the appropriate TROI console message).

Visually, when the VOI first starts up, the user will see the top level, base control panel for the VOI X interface (see Figure 8). It is through this control panel that the user is given access to all the other functions of the VOI. The top five buttons in the control panel, when pushed, pop up other frames which access the VOI's sub-functions[10]. Each of these sub-windows will be discussed in greater below.

The *Diagnostics* menu button allows the user to dump a representation of VOI's internal data structures to ***stdout***[11]. Under normal circumstances, the user will probably not need to use the diagnostic feature at all, and the information can help confirm DDD setup and other system mechanisms. This feature is present mostly for system diagnostic purposes.

The button entitled *Use Default Wakeup Interval* is used to set the event interlace interval to a known, non-antisocial setting. This button becomes necessary if the user sets the interlace interval to an unacceptable value (see Section 4.5).

10. The ellipses in these buttons are an OPEN LOOK convention for indicating that a dialog window will be popped open by pressing the button.
11. The triangle glyph at the right edge of this button is another OPEN LOOK convention indicating that a pull right menu is attached to the button. Pull down menu buttons are indicated with an arrow glyph pointing down.

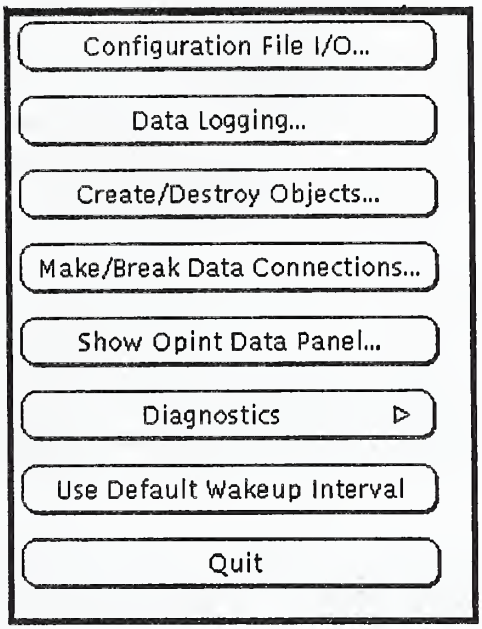

Figure 8. Opint Base Control Panel

The *Quit* button does the obvious thing: it exits the TROI program.

3.4.1 Visual Interface Manipulation: Operator Panel

The operator panel is the frame on which most, but not all, user interactions will occur once the configuration setup is determined, since most of the vo's in a configuration are placed within this frame. This includes sliders, pulldown menus, and text and numeric input and output. Figure 7 shows the entire terminal screen of a hypothetical VOI session, after some objects have been created and a data flow network has been determined. The frame entitled *Opint: Operator Panel* in the top half of the screen is the operator panel with a number of vo's placed on the panel

Vo's within the operator panel can be repositioned within the extent of the panel, and the panel itself can be resized and repositioned[12]. Vo positioning is accomplished by ***unlocking*** the visual object, then grabbing it with the middle mouse button and dragging the object to its new location. You can tell which vo you have selected by the highlight box drawn around the vo. Initially, all vo's are locked in their default position; selecting a vo with the middle mouse button while it is locked will not highlight the vo. Also, many vo's are aggregates of sub-components and these sub-components can be repositioned individually. To reposition sub-components, the vo must be ***uncoupled***. Re-coupling the sub-components will allow you to move the vo as a whole, with sub-components maintaining their relative positions. Vo's cannot be placed on top of each other and any new position which causes the currently selected vo to be placed on top of another vo will be disallowed. Controls for locking/unlocking vo's and for coupling/uncoupling vo sub-components are accessible from the *Objects Frame* discussed below.

When VOI first starts up, the operator panel will not be visible. By pressing the *Show Opint Data Panel...* button in the base control window, the user can pop open the operator panel. The operator panel cannot be dismissed, but it can be hidden or iconified[13]. The operator panel need not be visible to create or destroy vo's, although typically it is desirable to make this window visible at the start of the VOI session. This panel is automatically popped open when a new

12. The actual method of resizing and repositioning the panel depends on which X window manager is being used, and how the window manager has been configured. The X 11.4 ***twm*** window manager provides several methods for frame resizing and repositioning.
13. Dismissing a frame is different than iconifying the frame. When a frame is dismissed it is, in a sense, destroyed, to be recreated later when it is popped up again. Iconifying a frame does not destroy it, though what actually happens to the window depends on the X window manager being used, and how that window manager is configured. With the X 11.4 ***twm*** window manager, the frame may either be unmapped or iconified.

configuration is loaded from file.

When vo's are present on the configuration window the user may interact with these objects in OPEN LOOK compliant ways, which can usually be intuitively determined. For example, moving the cursor over the slider handle, pushing down the left mouse button, and dragging the handle left and right, changes the slider value. In general, the left mouse button is used to make a primary selection, and the right mouse button is used to pull up menus. The middle button is reserved for repositioning vo's within the operator frame. Section 2.2 and Appendix B:covers the specifics of the available vo's, how you can interact with them and what attributes they own.

3.4.2 Visual Interface Manipulation: Saving and Loading Configurations

Once a configuration has been set up, it can be saved to a file and reloaded during a future session, or at a later time in the current session. This way, you can have multiple data flows and interfaces corresponding to different sub-tasks. Configurations can be merged by loading one setup file after another, or you can clear the current configuration before loading the new one. All information pertaining to the current configuration is saved in the setup file, from vo positions to data flow connections. Only system variables as supplied by RCS are not saved in the configuration file since these are static and their existence depends on the state of the RCS system dictionary at the beginning of the VOI session.

The Configuration frame is used to load and store configurations ().This frame contains four buttons and a text input item. The file which will be loaded or stored is typed into the *Setup File* text input item. Standard relative or absolute UNIX[14] pathnames are valid here and some emacs-like editing control keys can be used inside the text field[6][11][12][15].

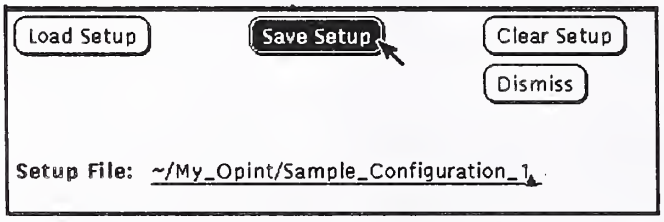

Figure 9. Configuration I/O Frame

The *Load Setup* button is used to load a configuration setup file. If no configuration is currently loaded, no confirmation is requested before.VOI attempts to open the configuration file specified for loading. If a configuration is already in use, then a confirmation is requested before the file is opened. Loading a setup file when VOI is already running a configuration will merge the two configuration setups. The merged configuration then becomes the current configuration.

Similarly, pressing the *Save Setup* button saves the current configuration into the file specified by the *Setup File* text input field. Figure 9 illustrates the invocation of the save operation. If the file already exists, then confirmation is requested before that file is over/written. Figure 10 illustrates what you would see if the save required confirmation. In this figure, the user is selecting to confirm the save.

If there are any problems in accessing the configuration file, either because the proper UNIX permissions are not set on the file or directory, or the file could not be found or opened, an error alarm box will appear, informing you of the problem.

As mentioned earlier, multiple configurations can be merged by simply loading one configuration file after the other. However, since object names must be unique, any subsequently loaded objects with duplicate names are ignored. An error alarm box will pop up, informing you when an object is ignored. If you do not want to merge a configuration,

14. UNIX is a trademark of AT&T.
15. Depending on the window manager in use, simply positioning the cursor over the text field may not be sufficient to activate the field for character input; you may need to click on the text field with the left button to activate the field. A grey diamond text cursor indicates the field is not accepting character input, while a black diamond text cursor indicates the field is accepting character input. This is S.O.P. for OPEN LOOK applications

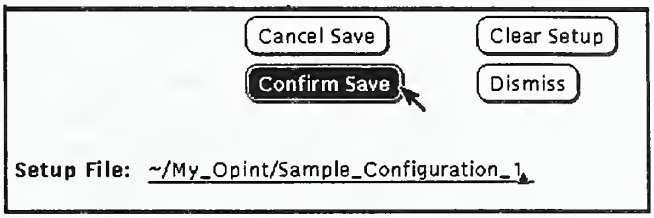

Figure 10. Confirming File Overwrite during Save

but instead overload a new configuration, first clear the current configuration using the *Clear Setup* button in this frame, before loading the new configuration file.

Almost the entire state of the VOI is saved in the configuration file. All objects (except system and internal objects) and their current attribute settings are saved, as are all variable connections, and the size of the operator panel. When loading a previously saved configuration, some parameters may no longer be valid. For example, if in a previously saved configuration, a connection was made to a RCS system variable which is not present during the current VOI session, this connection would no longer be valid. In these cases, the VOI will signal the problem to the user through the error Alarm Box. VOI will continue loading the configuration if possible.

The *Dismiss* button will hide the configuration frame until the *Configuration File I/O...* button in the main control panel is pushed again. You will find this *Dismiss* button in all the pop up dialog frames, and its function, as described previously, is consistent in all frames.

3.4.3 Visual Interface Manipulation: Visual Objects

In this section we will discuss how to create and destroy objects, both visual and non-visual, and how to selectively position vo's. Facilities which are available to allow the user to control attributes of objects will be covered in sections below.

The Object Frame (shown in Figure 11), is used to create and destroy visual and non-visual objects. This frame is divided into three subpanels: the top panel is used to name and create new objects, the center panel is used to destroy already existing objects, and the bottom panel is used to control positioning of visual objects in the Operator's frame.

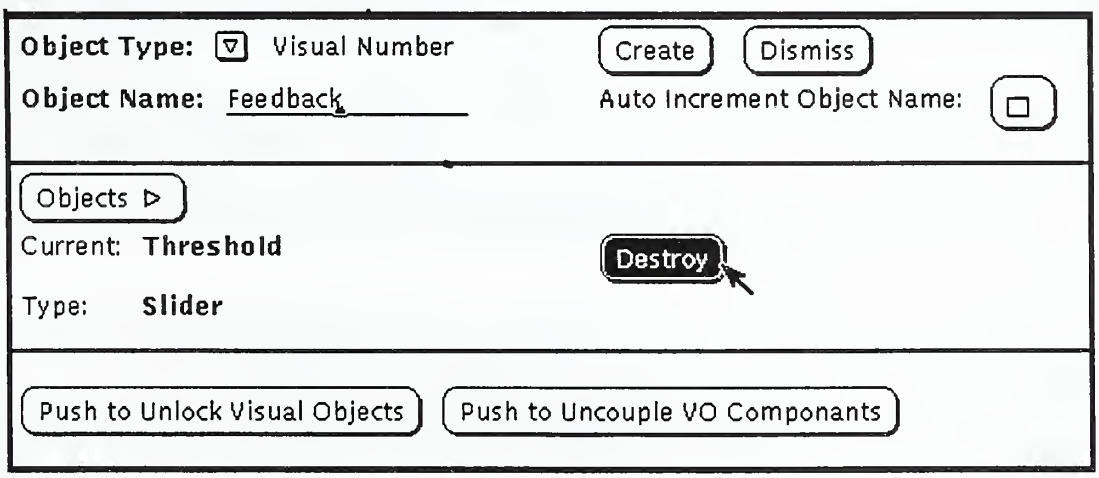

Figure 11. Object Frame

3.4.3.1 Visual Interface Manipulation: Creating Visual Objects

To create a new object, you must specify the object's name and the type of object to create. The object type is selected through the pulldown menu entitled *Object Type* at the top of the Create subpanel. The text to the right of the arrow

glyph reflects the currently selected type. Clicking the right mouse button over the arrow glyph or the object type string will pull down a menu presenting all the object types currently implemented.

Below the *Object Type* menu is a text field where you must enter the object's name. As mentioned previously, all objects in the system must be uniquely named. If an object with the current name already exists, the error Alarm Box will pop up to inform you of this situation, and the object will not be created. Object names have some side effects that you should note. For objects with visual components, the object name is often used to construct a default label on the visual elements, and to look up resources in the resource database (see Appendix B). For this reason, you should avoid using certain non-alphanumeric characters in the object names. If you need to have characters of this type in the object's label, you can use resources, described below, to set the object's label separately.

When the *Create* button is pressed, an object of the selected type and name is created. If no error message is displayed, you can assume the object was successfully created. Often, when an object is created, its visual components become visible on the screen, however some objects do not have any visible components, and others do not appear until some system state is reached. You can verify the object's existence by popping up the *Objects* menu (described below); if the newly created object's name appears in the list, then it has been successfully created.

If you are quickly prototyping a configuration and do not care what the exact object names will be, you can reduce the amount of typing needed to provide unique object names by using the ***auto-increment*** feature. Turning auto- increment on (by clicking on the box button to the right of the label *Auto Increment Object Name* so that a check appears in the box), will cause the system to insert a unique object name into the *Object Name* field whenever an object is created. Auto-increment will modify the current object name by incrementing a qualifier number suffix. This will guarantee that the object is uniquely named.

3.4.3.2 Visual Interface Manipulation: Destroying Visual Objects

Objects which have been previously created can be destroyed at any time. To destroy an object using this frame, you must first selected it from the alphabetical list of existing objects. You do this by pulling right the menu associated with the *Objects* button. When you click and hold the right mouse button over the *Objects* button, a menu will pop up containing a list of objects which are candidates for destruction. Some objects, specifically system objects associated with RCS system dictionary, cannot be destroyed and so would not be present in this list. Once an object is selected, the *Current* field will display the selected object's name and the *Type* field will display its type. Clicking on the *Destroy* button will destroy the selected object, however, confirmation will be required. If the object is successfully destroyed, the *Current* and *Type* fields will display the string "*none selected*" indicating that no object is currently selected and the object will no longer be present in the *Objects* list. Finally, the name of the object just destroyed is freed and can be used to name any future objects.

Destroying a graphical object will also destroy all the VOI graphical variables attached to the object. Destruction will also break all switchboard data flow connections to or from any of those destroyed variables. The system defined DDD entries remain accessible.

There is an alternate method for destruction of vo's within the Operator's Panel. If you grab a vo in this panel, it is possible to drag it outside the extent of the panel. If you release your grab on the vo while it is fully outside the visible extent of the panel, the vo will be destroyed.

3.4.3.3 Visual Interface Manipulation: Locking and Unlocking Visual Objects

As mentioned above, vo's in the Operator's Panel are locked in their default positions when created and must be unlocked before they can be repositioned. Also, most vo's are aggregates of sub-components and these sub-components must be uncoupled before they can be independently repositioned. There are two binary (two-state) buttons in the bottom sub-panel which control locking/unlocking of vo's and coupling/uncoupling of vo sub-components.

When vo's are locked in position, the lock/unlock binary button's label will read *Push to Unlock Visual Objects*. Pushing the button will unlock vo's allowing you to grab them with the middle mouse button and directly manipulate their positions as aggregate objects. When vo's are unlocked, the binary button's label will read *Push to Lock Visual Objects*. Pushing the button at this time will lock vo's in their current positions.

The couple/uncouple binary button works in analogous manner. When the couple/uncouple button's label reads *Push*

to Uncouple vo components, pushing the button allows you to grab sub-components with the middle mouse button, and directly manipulate their positions relative to each other. Pushing the button when its label reads *Push to Couple vo components*, re-couples vo sub-components, allowing you to directly manipulate vo aggregate objects. Note that vo's must be unlocked before you can manipulate sub-component positions.

vo's and vo subcomponents cannot be positioned on top of each other by the user. When directly manipulating vo or vo component positions, the object being moved about may at times freeze in place if its commanded position would overlap another object. Direct manipulation of the grabbed object's position will resume when you move the cursor outside the extent of the other object.

3.4.4 Visual Interface Manipulation: Configuring Data Flow Connections

At any time during your configuration set up, you may ***make*** or ***break*** data flow connections from producer variables to consumer variables. By making and breaking data flow connections, you will set up data paths which will propagate information from the user, through visual objects, possibly through data filters, and finally to RCS. Similarly you will set up data paths which will enable information to flow from RCS, eventually reaching a vo which graphically displays this information.

3.4.4.1 Visual Interface (VOI) Manipulation: Making and Breaking Connections

The Connections frame (Figure 12) is used to make and break connections between VOI dictionary variables. This frame has two sub-panels: the top panel is used to make data flow connections and the bottom panel is used to break data flow connections.

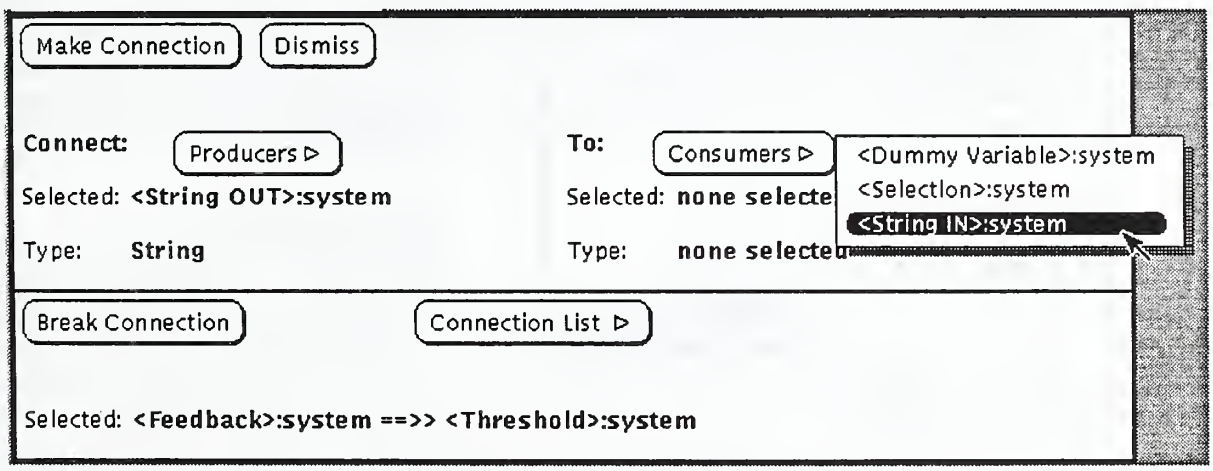

Figure 12. Connections Frame

Before a connection can be made, both ends of the data connection must be selected. You do this by popping up the menus attached to the *Producers* menu button and the *Consumers* menu button, and selecting a variable for each terminus of the connection. Only the variables which are allowed to be producers are shown in the producers list, similarly for consumers. The selected producer's name and variable type will be displayed below the *Producers* menu button; likewise for the selected consumer. In Figure 12, the user is in the middle of the selection process. A producer variable has already been selected and the user is about to select the consumer variable for the other end of the connection.

Variables of different type cannot be connected together and a variable of an undetermined direction cannot be connected to itself. Once both a consumer and a producer are selected, pushing the *Make Connection* button will attempt to attach the two variables. If the connection is unsuccessful, an error message will be displayed to indicate the problem. If the connection was successful, the consumer will be removed from the *Consumers* list[16], and the consumer's *Selected* and *Type* fields will revert to "*none selected*" to indicate no consumer is currently selected.

To break a connection, you first select a connection from the *Connection List* pull right menu in the lower sub-panel, then push the *Break Connection* button. Connections are displayed in the following format: "*<producer>* ==>> *<consumer>*", where *<producer>* represents the fully scoped name of the producer terminus and *<consumer>* represents the fully scoped name of the consumer terminus.

When a connection is successfully broken, the selected connection message reverts to "*none selected*", and both variables are recycled into the appropriate pool of candidate producers and/or consumers so that they can later be re-connected to form a new data flow.

As soon as a data flow connection is made, information can begin to flow along that connection. If the producer variable contains a value before it is connected, that value will be propagated as soon as a connection is made. Thus, since some vo's can be given default values, these values can be sent along their path to RCS automatically by making the appropriate data flow connection.

3.4.5 Visual Interface Manipulation: Data Logging

System variables supplied by RCS can be ***data logged*** to a file for examination. The RCS data server process handles the actual data logging operations for which the VOI provides a convenient front-end. These functions are accessed through the *Data Logging* frame as shown in Figure 13. To perform data logging on a system variable, you must first select the variable to log on by choosing one from the alphabetical list presented in the *System Variables* pull right menu. Figure 13 shows the user in the process of selecting the *"Threshold"* system variable for data logging. The currently selected variable will be displayed in the *Selected* message field. Data logging operations can be performed on only one variable at a time, but multiple log files can be open simultaneously, and logging can be on-going for more than one variable at a time.

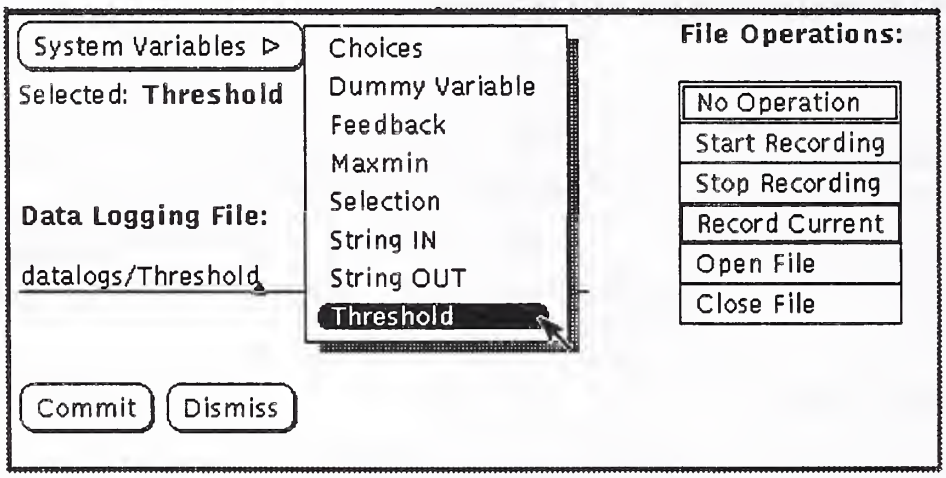

Figure 13. Data Logging Frame

The name of the data logging file to which values will be stored is entered in the *Data Logging File* text input field. This file name is a standard UNIX path. The file operation to perform on the selected system variable is chosen from the *File Operations* menu. Only one operation can be selected at a time. The available operations are described as follows:

1. No Operation
   This is the null file operation.
2. Start Recording
   Informs RCS to begin recording a data series of the selected variable. Data logging will continue on this

16. Since the consumer has just been attached to a producer, and consumers may only consume from a single producer, this variable is no longer a candidate for future connections, until this connection is first broken.

variable until a *stop recording* operation is performed.

3. **Stop Recording**
   Informs RCS to stop a data series recording. Used in conjunction with the *start recording* operation.
4. **Record Current**
   Inform RCS to record only the current value of the selected variable.
5. **Open File**
   Informs RCS to open the data logging file as specified in the *Data Logging File* text input field. This must be performed before any data logging can commence to the file.
6. **Close File**
   Closes the selected data logging file previously opened with an *open file* operation.

Making a selection from the *File Operations* menu does not institute the chosen file operation. The user must click on the *Commit* button for the file operation to take effect. Also, the data logging frame remembers the last file operation committed on a variable.

4 Customizing a TROI System

TROI divides the user-interface into two components, the data server supplying the real-time target system connection and the visual interface system. The target system data server and the visual interface communicate via the data dictionary. The ***fundamental TROI activity*** is defining the data dictionary that connects the real-time system and the visual interface. Once the data dictionary has been defined, compiled, and linked, then the user constructs the actual visual interface interactively. A set of scripts and templates within a sample program are available to simplify building and defining the data dictionary. Customizing a TROI application system can be greatly simplified by using the sample template programs and following a set of step by step directions. The last sections under customizing involve personalizing the TROI visual interface with the use of the X resource manager. Again, it is helpful if one is familiar with X.

4.1 Customizing DDD: Step by Step

Customizing a TROI system depends on the user defining the important RCS variables that must be available for the user to manipulate. Following the directions to install and generate the sample explanatory TROI session will provide a good foundation for customizing a TROI system. This explanatory TROI session contains template files that include comments about substituting different code to generate a different TROI application. The sample template files include: *data_defintions_a*, *data_definitions.a*, *simulator.a* and *oi.a*. (Refer to Appendix C for listing of these files.) Within these files are comments which outline, step-by-step, the necessary instruction to building a TROI system. In addition, these files illustrate concepts with simple examples.

4.1.1 Customizing TROI: Data Type Definitions Specification

The ***TROI data definitions specification*** is the first step in constructing the DDD. The basic purpose in this process is to define the data types. The bulk of the type definitions occur in the file *data_defintions_a*. The data typing specification is available to both the main oi procedures and the simulator. The following steps must be performed within the file *data_definitions_.a*:

1. Define the `READER/WRITER` buffer names.
2. Define the application specific commands and status types for defining the `READER/WRITER` buffers.
3. Define boundary ranges of variables.
4. Define the generic instantiation of the reader/writer buffers. Note: The TROI version of the reader writer does not allow typed naming of each reader/writer buffer. Hence, there is no `NAME` field in the generic definition.
5. Define the local working copies of the global reader/writer buffers.

6. Provide an init procedure in body to initialize reader/writer buffers types for defining the `READER/WRITER` buffers.
7. Provide any specialized callback functions.

In customizing, the user could replace or augment this package with existing application-specific data packages.

4.1.2 Customizing TROI: Data Type Definitions Body

The ***TROI data definitions body*** is the second step in constructing the DDD. The basic purpose of this step is to initialize the physical location and values of the reader/writer buffers. The following steps must be performed within the file *data_definitions.a*:

1. Initialize local reader/writer buffers.
2. Reset the physical location of reader/writer instantiations. This implies supplying a base address for the reader/writer buffer, read in from the `buffer_addresses` input file.

   Note: `buffer_addr` is an array of indexed by integers, not indexed via enumerated types as with previous versions. The change was necessary to allow a more general `buffer_addresses` specification. Previously, changing the `buffer_addresses` spec. required recompile of numerous other dependent r/w files.

   ```
   RW_PARAMETER.RESET( BUFFER_ADDRESSES.buffer_addr (to_int(RW_PARAMETER_BUFFER)));
   ```

3. Initialize the global copy of the r/w buffer. If un-initialized already, the local copy of the r/w buffer will be written to the global r/w buffer.
4. Supply specialized callback functions.

4.1.3 Customizing TROI: Data Description Definitions in *oi.a*

The ***TROI data description definitions*** is the third step in constructing the DDD. The basic purpose of this step is to initialize the physical location and values of the reader/writer buffers. The following steps must be performed within the file *data_definitions.a*:

1. `WITH` the Operator Interface package: `buffer_addresses`, `oi_globals`, `oi_rcs`, `oi_x`, and `oi_x_dict` and User Supplied Data Definition & simulator packages
2. Include generic nop callback to allow compiler to find. Always include:

   ```
   Procedure callback renames oi_x.callback;
   ```

3. Instantiate new `generic buffer_addresses` with enumerated type via file reading .

   When invoked, the routine will read the *buffer_addresses.dat* file.

   ```
   procedure SAMPLE_BUFFER_INIT is new BUFFER_ADDRESSES.INIT(ELEMENTS => BUFFER_NAMES);
   ```

4. Data routing definitions using generic instantiation. Example:

```
package OI_SAMPLE_COMMAND_ECHO is new OI_X.MENU_BUILD
    (
    NAME_IN              => "Command Selection",
    NAME_OUT             => "Command List",
    VAR_ADDR             => LOCAL_COMMAND.MODE'ADDRESS,
    ENUM                 => PROCESSING_MODES,
    UPDATE               => 10,

    RW_ENTRY             => TRUE,
    RW_NAME              => "RW_COMMAND",
    RW_SIZE              => COMMAND_TYPE'SIZE,
    RW_LOC_BUFFER_ADDR => LOCAL_COMMAND'ADDRESS,

    FILE_NAME            => "command_trace.dat",
    ENTRY_USE            => OI_X.SHOW
    );
```

5. After the *begin* is the run-time code. This code contains standard TROI code, plus any new run-time DDD definitions, and any run-time initialization of the physical addressing of reader-writer buffers. Revamp remaining TROI code to reflect new packaging (wherever sample, replace with new TROI name). And add the run-time DDD definitions.

```
OI_COMMAND_LINE.PARSE; -- parse the command line run time parameters
                       -- initialize r/w buffers memory locations
SAMPLE_BUFFER_INIT;
DATA_DEFINITIONS.INIT; -- initialize opint command buffers, etc.
                       -- user initializes local buffers before sending
                       -- out image to X
OI_SAMPLE_COMMAND_ECHO.INIT(BUFFER_ADDR(to_int(RW_COMMAND_BUFFER)));
if(SIMULATOR)          -- start up simulator with signal
    then SAMPLE_SIMULATOR.SIMULATOR_SYSTEM.startup;
    end if;
OI._X.PROCESS;         -- when user session is complete, return to next line
if(simulator) then     -- specialized program termination
    abort PRIM_SIMULATOR.SIMULATOR_SYSTEM;
end if;
```

4.2 Running the new TROI package:

After compiling and linking the newly customized TROI package, it is time to run a TROI system. The command line options available that are embedded with TROI are:

```
    -self              : choose r/w buffers on self
    -simulator         : enable the simulator to operate
    -display x:0.0     : select X server display name
                          in this case x is a host name e.g. daneel, giskard...
    -debug             : show underlying actions (messy)
    -semaphore         : enable semaphoring (Sun VME tas instruction)
    -buffer filename   : the r/w buffer addresses file
                          and any X specific parameters that are passed to
                          the X interface.
```

When testing out an application with the simulator, the command line options ***-self*** and ***-simulator*** are both selected.

4.2.1 Resources and Command Line Switches

All vo's have various behaviors and attributes that can be changed by setting the appropriate ***resource value*** controlling the behavior. Each vo defines a set of resources that it will recognize, and the VOI application itself recognizes resources controlling the overall behavior of the application. VOI utilizes standard X window[17] mechanisms for specifying ***resource settings***. This section will describe resources in a very general sense. For more information see [9][16][13][6].

An X resource setting consists of two parts, the resource ***specification*** (i.e., the resource that is going to be set), and the ***value*** for that resource. Resources settings are usually grouped together in a ***resource file***, with a single setting on each line. The resource specification is separated from the value by a colon (":") and optional white space. The end-of-line character separates one resource setting from another. For example, if an application understood two resources, *foo* and *bar*, each of which took an integer value, you might have the following lines in your resource file:

```
foo: 7
bar: 8
```

In a UNIX command shell the resource file is loaded into the X server process with the *xrdb* program like so:

```
% xrdb <my_defaults_file>
```

Full resource specifications are composed of a hierarchy of resource specifications, with each level in the hierarchy separated by a period (".") or asterisk ("*"). An asterisk is a wildcard resource specifier, allowing portions of the resource hierarchy to be omitted. The resource hierarchy of an X application is specified by the application; illustrates VOI's resource hierarchy. Resources relating to vo attributes are discussed in greater detail in Appendix B:.

Resources associated with the VOI itself, are also accessible through command line switches in the traditional UNIX conventions. Here is a list of the command line switches recognized by VOI.

- **-go** set the ***load-and-go*** flag. When this is set, VOI automatically loads the default configuration file (either the file named "*Configuration*" in the current directory, or the file specified by the **-f** switch below) when it starts up.
- **-sec** set the interlace interval seconds value.
- **-usec** set the interlace interval milliseconds value.
- **-f** set the default configuration file name for loading and storing.
- **-rn** set the resource name for the VOI application. The resource name will be used to find resources specific to this instance of the VOI application[18].
- **-xrm** supply an X resource setting on the command line[19].

More familiarity with the use of X resource settings and standard X application conventions may be required to gain full benefit of these switches in particular, and the resource setting model for controlling vo behavior in general.

4.3 Object Resources

Most VOI objects are aggregates of visual components based on widgets in the XView toolkit [6]. As is the convention with X based applications, the VOI recognizes a set of resources which control the specific behaviors of the various visual objects. Appendix B: outlines the Resource Hierarchy for visual objects. The sections below discuss the resources recognized by the objects in greater detail. In this discussion, the ***resource class*** of the object is given (where appropriate). These classes are the same for every object of that type and can be used to set the behavior of all

17. Version 11, release 4.
18. This switch is used to specify a string which VOI will use to match the highest level resource specification. In this way, the user can have more than one set of resource specifications for different invocations of VOI. [9][16][13]
19. This is a convenience. A usage example might be: `opint -xrm "foo: 7"`.

objects of a designated type. Objects will also have a ***resource name*** which differs for each instance of the object. The object's resource name is the name given to the object by the user when it is created.

4.3.1 Alarm Box Object

The alarm box object has a resource class of ***AlarmBox***, and recognizes two resources. The first has a resource name of ***geometry***, resource class ***Geometry***. The alarm box only recognizes the positional elements of the geometry specification; the size of the box is determined by the length of the string message. The second resource is named ***label***, resource class ***Label*** used to set the frame's label string.

4.3.2 Cursor Tracker Object

The cursor tracker object has a resource class of ***CursorTracker*** and recognizes three resources. The ***geometry*** and ***label*** resources are similar to those of the alarm box object. The third resource controls the cursor tracking method described in section 2.2.2. Its resource name is ***alwaysTrack*** and its resource class is ***AlwaysTrack***. Default is false, indicating operation in *Button Down Only* mode. Setting to true indicates operation in *All Cursor Motion* mode.

4.3.3 Input String

The input string object has a resource class of ***InputString*** and like other objects, recognizes a ***label*** resource. It also recognizes the resource named ***fieldLength***, resource class ***FieldLength***, which is the number of characters that can be typed into the object's text input field. The input string object also recognizes an initialization string resource called ***initializationValue***, resource class ***InitializationValue*** which you can use to set the default string value for the object.

4.3.4 Pulldown Menu

Pulldown menus have a resource class of ***PulldownMenu*** and recognize only the ***label*** resource.

4.3.5 Slider

The slider has a resource class of ***Slider*** and recognizes eight resources. As with other objects, the slider reads the ***label*** resource, and as with input string objects the slider reads the initialization value resource ***initializationValue***, though with the slider, this value should be a floating point number. The slider recognizes three boolean resources: 1) the ***vertical*** resource, class ***Vertical***, which, when set to true will make the slider orient vertically instead of the default horizontal orientation[20]; 2) the ***integer*** resource, class ***Integer***, which, when set to true, forces the slider to input or output only integers instead of the default double precision, floating point numbers[21]; and 3) the ***dynamicRange*** resource, class ***DynamicRange***. This resource determines whether the range (maximum and minimum) can be set dynamically (i.e., via RCS) or is static for the life of the slider. When this resource is set to true (the default), the *range* variable is created and when a new value appears in this consumer, the slider's maximum and minimum will change. In this case any ***maximum*** and ***minimum*** resource settings are taken as default values for the range. When the ***dynamicRange*** resource is set to false, the *range* variable will not be created and the ***maximum*** and ***minimum*** resources are taken as the static range values.

As implied above, the slider recognizes the range resources ***minimum*** (class ***Minimum***), and ***maximum*** (class ***Maximum***). The slider also recognizes a resource describing the width of the slider in pixels. This resource is named ***width***, class ***Width***. By setting the ***width***, ***maximum***, and ***minimum*** resources you can control the scaling factor used

20. Vertical sliders are currently not implemented in XView.
21. The value consumed or produced is still a double, but it is rounded to the nearest integer value.

to convert slider handle positions to values within the slider's range. The scaling formula is:

$$\frac{(\text{value} - \text{minimum}) \times \text{width}}{\text{maximum} - \text{minimum}} = \text{handle}$$

where *value* is the value of the slider's consumer (or producer) variable; *maximum*, *minimum*, and *width* are the user or system supplied resources described above; and *handle* is the position of the slider handle within the slider body.

4.3.6 Visual Number

The visual number object has a resource class of ***VisualNumber*** and recognizes only the ***label*** resource.

4.3.7 Visual String

The visual string object has a resource class of ***VisualString*** and recognizes the ***label*** resource.

4.4 Example Resource Settings

The following snippet from a resource file provides example resources settings which would be appropriate for controlling various attributes. Note that the exclamation point denotes a commented line.

```
! Opint resources
!
Opint.loadAndGo:True
Opint.baseFrame.geometry:+0+0
!
! Slider resource settings
!
Opint.Slider.dynamicRange:False
Opint.Slider.integer:True
Opint.Slider.minimum:-180.0
Opint.Slider.maximum:180.0
Opint.Slider.width:90
!
! threshold instance settings
!
Opint.threshold.minimum:0
Opint.threshold.maximum:255
Opint.threshold.width:512
```

To explain this example resource file, suppose the interface being used looks somewhat likeFigure14, where there are 3 sliders, two of which control joint angles, and a third which controls a threshold value.

With the resource file above, the user is specifying some application-wide behaviors such as load-and-go and the position on the X screen of the *baseFrame* object[22]. The user is also specifying that all objects of class *Slider* display only integer values, have no dynamic range, have a maximum of 180.0 and a minimum of -180.0 and be 90 pixels wide. However the user has also specified that a particular instance of slider (named *threshold* by the user), override the class values for minimum, maximum, and width in favor of the instance values of 0, 255, and 512 respectively.

4.5 The Interval Timer

Because VOI is both window event driven and data event driven, it must merge events from two separate streams. To accomplish this, VOI's processing loop interlaces window processing and data flow processing. The window cycle is

22. This position is supplied as a user hint to the window manager, which may or may not honor such a hint.

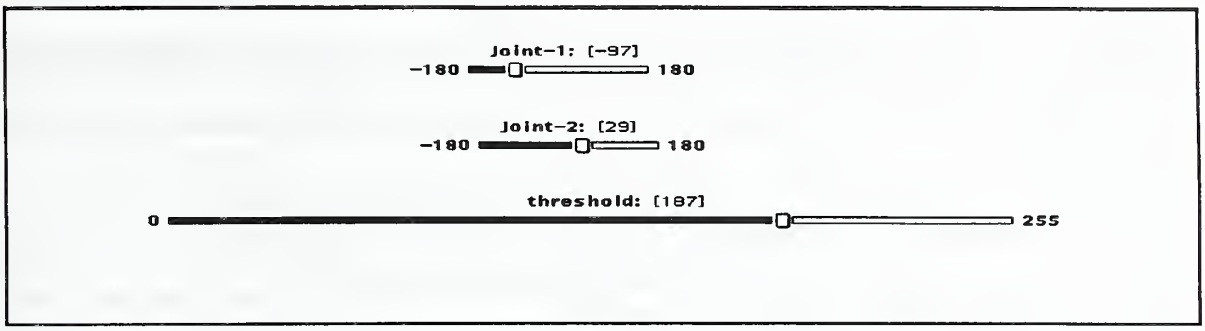

Figure 14. Resource Example

a standard XView notifier loop [6] which spends a certain amount of time processing X events before breaking out and returning control to the application (in this case the VOI data flow processing cycle). The amount of time VOI spends in the X window event cycle before processing data events is called the ***interlace interval*** and is user configurable. The user can specify (via command line switches or resources -- both previously covered), the interval in seconds and microseconds, however, the effective granularity of the window event loop is around 30 milliseconds (i.e., 30000 microseconds). Thus interval specifications less than this granularity (but non-zero) will default to an interval which processes data events as quickly as possible. As the user increases the amount of time spent in the window event loop, VOI becomes more responsive to user interactions, but less responsive to changes in the data (see Figure 15). The responsiveness of the running system is hard to determine in a general case since it will depend on secondary factors such as network traffic, system load, the number of open X applications connected to your server, and the amount of data being processed.

Setting the interval to zero seconds and zero microseconds is a special case which turns off the data processing loop. Thus, at this point, VOI would be as responsive as possible to user interactions, but would never display changes in the RCS to the user. For this reason the *Use Default Wakeup Interval* button is provided in the main control panel. If you accidently set the interval to zero, pushing this button will reset VOI to process data events as fast as possible.

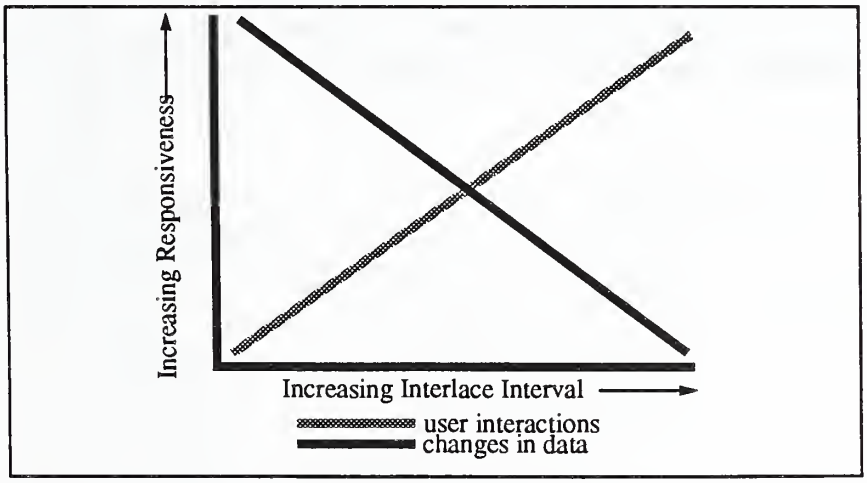

Figure 15. X Responsiveness vs. Increasing Interlace Interval

Appendix A: References

[1] Albus, James S., Harry G. McCain, Ronald Lumia, "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)", NIST Technical Note 1235, 1989 Edition.

[2] Booche, Grady. *Software Engineering with Ada, Second Edition*, Benjamin/Cummings Publishing, Inc. Menlo Park, Cal., 1987.

[3] Brinch Hansen, Per. *Operating System Principles*. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1973.

[4] DoD *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983.

[5] Fiala, J.C. "Note on NASREM Implementation" NIST Internal Report 89-4215, National Institute of Standards and Technology, Gaithersburg, Md., December 1989.

[6] Heller, Dan, *XView Programming Manual, An OPEN LOOK Toolkit for X11*, O'Reilly & Associates, Inc., Volume 7, 1989.

[7] Hoare, C.A.R. "Monitors:An Operating System Concept, Communications of the ACM", Vol. 17, No. 10, 1974.

[8] Kernighan, Brian W., Dennis M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[9] Nye, Adrian, *Xlib Programming Manual for Version 11*, O'Reilly & Associates, Inc., Volume 1, May 1989.

[10] Mandelkern, D. "A GUIDE to High-Level User Interface Development Tools," SUN Expert Magazine, Vol. 1, Num. 3, Jan. 1990.

[11] *OPEN LOOK Graphical User Interface Specification, Release 1.0.1*, Sun Microsystems, Inc., August 1989.

[12] *OPEN LOOK User Interface Style Guide*, Sun Microsystems, Inc., August 1989.

[13] O'Reilly, Tim, Valerie Quercia, Linda Lamb, *X Window System User's Guide for Version 11*, O'Reilly & Associates, Inc., Volume 3, 1988.

[14] *VADS: VERDIX Ada Development System User's Guide*, Verdix Corporation, Chantilly, Va. 1989.

[15] *VADS: VERDIX Ada Development System Programmers Guide*, Verdix Corporation, Chantilly, Va. 1989.

[16] *Xlib Reference Manual for Version 11*, O'Reilly & Associates, Inc., Volume 2, May 1989.

Appendix B: TROI Resource Management

Resource Name (**ResourceClass**)	Legal Values	Default Value
<applicationResourceName>[1] (**Opint**)		
loadAndGo (**LoadAndGo**)[2]	True/False	False
seconds (**Seconds**)[3]	*<integer>*	0
microseconds (**Microseconds**)[4]	*<integer>*	350
baseFrame (**Frame**)		
geometry (**Geometry**)	*<X geometry specification>*[5]	*<not set>*
connectionsFrame (**Frame**)		
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
datalogFrame (**Frame**)		
filename (**Filename**)		
value (**Value**)	*<UNIX path>*	./log.dat
fieldLength (**FieldLength**)	*<integer>*	35
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
configurationFrame (**Frame**)		
filename (**Filename**)		
value (**Value**)[6]	*<UNIX path>*	./Configuratio
fieldLength (**FieldLength**)	*<integer>*	30
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
confirm (**Confirm**)	True/False	True
objectsFrame (**Frame**)		
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
autoIncrement (**AutoIncrement**)	True/False	False
confirm (**Confirm**)	True/False	True
operatorFrame (**Frame**)		
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
<objectName> (**AlarmBox**)		
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
label (**Label**)	*<string>*	*<objectName>*
[SystemAlarm] (**AlarmBox**)[7]		
<objectName> (**CursorTracker**)		
geometry (**Geometry**)	*<X geometry specification>*	*<not set>*
label (**Label**)	*<string>*	*<objectName>*
alwaysTrack (**AlwaysTrack**)	True/False	False
<objectName> (**InputString**)		
label (**Label**)	*<string>*	*<objectName>:*
fieldLength (**FieldLength**)	*<integer>*	25
initializationValue (**InitializationValue**)	*<string>*	*<not set>*
<objectName> (**PulldownMenu**)		
label (**Label**)	*<string>*	*<objectName>:*
<objectName> (**Slider**)		
label (**Label**)	*<string>*	*<objectName>:*
vertical (**Vertical**)	True/False	False
integer (**Integer**)	True/False	False
dynamicRange (**DynamicRange**)	True/False	True
minimum (**Minimum**)	*<double>*	-100.0
maximum (**Maximum**)	*<double>*	100.0
width (**Width**)	*<integer>*	100
initializationValue (**InitializationValue**)	*<double>*	*<not set>*
<objectName> (**VisualNumber**)		
label (**Label**)	*<string>*	*<objectName>:*
<objectName> (**VisualString**)		
label (**Label**)	*<string>*	*<objectName>:*

1. *<applicationName>* is either the program invocation name, or the name specified with **-rn** ***<applicationResourceName>***.
2. command line switch equivalent is **-go**.
3. command line switch equivalent is **-sec** ***<seconds>***.
4. command line switch equivalent is **-usec** ***<microseconds>***.
5. X geometry specifications have the following format: [=]*<width>*x*<height>*{+-}*<xoffset>*{+-}*<yoffset>*. See [9], [16].
6. command line switch equivalent is **-f** ***<filename>***.
7. The *[SystemAlarm]* object is created by the opint system to report errors to the operator.

Appendix C: Template Files

File: *data_definitions_.a* : Data Definitions Specification

```
-------------------------------------------------
-- HEADER : data_definitions_.a
-- This package contains sample data definitions.
--
-- The packages that must be included are

-- UNCHECKED_CONVERSION : for buffer_type to enumerated type conversion
-- READER_WRITER : for reader/writer buffer declarations
-------------------------------------------------
with UNCHECKED_CONVERSION;
with READER_WRITER;
package DATA_DEFINITIONS is
-------------------------------------------------
-- Section 1. Defining the READER/WRITER Buffer Names
-- This section will be similar in all oi definitions
-------------------------------------------------
-- buffer addresses interface!
 type buffer_names is
              (
              PAGE_BOUNDARY,
              RW_COMMAND_BUFFER,
              RW_STATUS_BUFFER,
              RW_PARAMETER_BUFFER

              );
                        -- use full word for representation
       for buffer_names'size use 32; -- simplifies life

       function to_int is new UNCHECKED_CONVERSION
       ( SOURCE => buffer_names, TARGET => integer);
-------------------------------------------
-- Section 2. Define the application specific commands/status
-- status types for defining the READER/WRITER buffers.
-------------------------------------------
 type PROCESSING_MODES is (SALUTING, AT_EASE, MARCHING);
       type COMMAND_TYPE is
        record
              COMMAND_NO          : INTEGER;
              MODE                : PROCESSING_MODES;
              SPEED               : DURATION;
              ENERGY              : SHORT_FLOAT;
       end record;

 type STATUS_MODES is (DONE, EXECUTING, ERROR);
 type ERROR_MODES is (OK,CONFUSED, DAZED, TIRED);

 type STATUS_TYPE is
        record
              ECHO_COMMAND_NO     : INTEGER;
              STATUS_MODE         : STATUS_MODES;
              ERROR_TYPE          : ERROR_MODES;
       end record;

 type JOINTS_TYPE is array(1..7) of short_float;

 type PARAMETER_TYPE is
       record
              X                   : INTEGER;      -- x position
              Y                   : INTEGER;      -- y position
              Z                   : INTEGER;      -- z position
              J                   : JOINTS_TYPE;  -- arm joint pos.
       end record;

-----------------------------------0-----------------
-- Section 3. Define ranges with boundaries of variables
-------------------------------------------------------
       type boundary_type is array (1..2) of float;
       TIMING_BOUNDS : boundary_type := ( 0.1, 2.0);
-------------------------------------------
-- Section 4. Define the generic instantiation of the
-- reader/writer buffers.
-- Note: The OI version of the reader writer does not allowing
-- typed naming of each reader/writer buffer. Hence, there
-- is no NAME field in the generic definition.
-------------------------------------------
 package RW_COMMAND is new READER_WRITER (DATA => COMMAND_TYPE);

 package RW_STATUS is new READER_WRITER (DATA => STATUS_TYPE);

 package RW_PARAMETERis new READER_WRITER (DATA => PARAMETER_TYPE);
-------------------------------------------------------
-- Section 5. Define the local working copies of the global reader/writer buffers.
-------------------------------------------------------
       LOCAL_COMMAND: COMMAND_TYPE;
       LOCAL_STATUS: STATUS_TYPE;
       LOCAL_PARAMETER: PARAMETER_TYPE;
-------------------------------------------------------
-- Section 6. Provide a init procedure in body to initialize
-- reader/writer buffers
-- types for defining the READER/WRITER buffers.
-------------------------------------------------------
 procedure INIT;          -- initialize data buffers used by OP
-------------------------------------------
-- Section 7. Provide any specialized callback functions.
-------------------------------------------
       procedure samplecallback ;
end DATA_DEFINITIONS;
```

```
File: data_definitions.a - Data Definitions Body
-------------------------------------------------
This package contains sample data definition body -
include: initializations and callbacks definitions.
-------------------------------------------------
with BUFFER_ADDRESSES; -- must be included
with TEXT_IO;                    -- included for sample output

package body DATA_DEFINITIONS is

procedure INIT is
-------------------------------------------
-- Section 1. Initialize local reader/writer buffers.
-------------------------------------------
begin
      LOCAL_COMMAND :=
                              (COMMAND_NO=> 0,
                               MODE=> AT_EASE,
                               SPEED=> 0.2,
                               ENERGY=> 10.0
                              );

      LOCAL_STATUS :=
                              (ECHO_COMMAND_NO=> 0,
                               STATUS_MODE=> DONE,
                               ERROR_TYPE=> OK
                              );

      LOCAL_PARAMETER :=
                              (X      => 0,
                               Y      => 0,
                               Z      => 0,
                               J      => ( 20.0, 30.0, 10.0, 45.0, 25.0, 45.0, 33.0)
                              );

-------------------------------------------
-- Section 2. Reset the physical location of reader/writer instantiations. This implies
-- supplying a base address for ther/w buffer, read in from the buffer_addresses input
-- file (see and sample_data's spec. main.a).
--
-- Note: buffer_addr is an array of indexed by integers, not indexed via enumerated types as
-- with previous versions. The change was necessary to allow a more general buffer_address
-- specification. Previously, changing the buffer_address spec. required recompile of
-- numerous other dependent r/w files.
-------------------------------------------
      RW_COMMAND.RESET(BUFFER_ADDRESSES.buffer_addr(to_-
int(RW_COMMAND_BUFFER)));
      RW_STATUS.RESET(BUFFER_ADDRESSES.buffer_addr(to_-
int(RW_STATUS_BUFFER)));
      RW_PARAMETER.RESET(BUFFER_ADDRESSES.buffer_addr(to_-
int(RW_PARAMETER_BUFFER)));

-------------------------------------------------------------------------
-- Section 3. Initialize global copy of r/w buffers.
-- If uninitialized already, the local copy of the r/w
-- buffer will be written to the global r/w buffer.
-------------------------------------------------------------
      RW_COMMAND.INITIALIZE( LOCAL_COMMAND );
      RW_STATUS.INITIALIZE( LOCAL_STATUS );
      RW_PARAMETER.INITIALIZE( LOCAL_PARAMETER );

end INIT;

-- -----------------------------------------------------------------------
-- Section 4. Supply specialized callback functions. This feature is best explained by
-- examples.
--
-- 1. Suppose you want to signal six different variables when one variable is changed.
-- Use the callback feature to modify the five other variables when the
-- initial variable changes.
-- 2. Suppose, you want to transform data from angles to radians, the callback could be used
-- to do this transformation.
-------------------------------------------------------------------------
procedure samplecallback is

 begin
      TEXT_IO.put_line("This demonstrates the ability for user OI customization");
end samplecallback;

begin

      null;

end DATA_DEFINITIONS;
```

```
File: simulator.a - Simulation Specification and Body
--
-- This is a simulator to exercise the sample data
-- definitions.
--
-------------------------------------------
-- ** WITH packages list:
-------------------------------------------
                                    -- SYSTEM SUPPLIED
with SYSTEM;            use SYSTEM;
with TEXT_IO;           use TEXT_IO;
                                    -- OI SUPPLIED
with OI_GLOBALS; use OI_GLOBALS;
with DATA_DEFINITIONS;use DATA_DEFINITIONS;

package SAMPLE_SIMULATOR is

task SIMULATOR_SYSTEM is
      pragma PRIORITY(14);
 entry startup;
end SIMULATOR_SYSTEM;

end SAMPLE_SIMULATOR;

package body SAMPLE_SIMULATOR is

package enum_io is new enumeration_io(PROCESSING_MODES);
package enum_status_io is new enumeration_io(STATUS_MODES);
package sim_int_io is new integer_io(integer);

-----------------------------------------------
--** Local declarations for Reader/writer----
-----------------------------------------------
SIMULATOR_COMMAND  : COMMAND_TYPE;
SIMULATOR_STATUS : STATUS_TYPE;
SIMULATOR_PARAMETER : PARAMETER_TYPE;

      LAST_COMMAND_NO : integer := -1;-- has the command number changed? if yes,

-----------------------------------------------
--** PIPE_SIMULATOR - simulate pipe level2
-----------------------------------------------
task body SIMULATOR_SYSTEM is
                                    -- indicates new command to work on.
begin
      accept startup;               -- wait for signal to start
      put("SAMPLE SIMULATOR STARTED"); new_line;
                                    -- initialize status
      SIMULATOR_STATUS.ECHO_COMMAND_NO:= 0;
      SIMULATOR_STATUS.STATUS_MODE := DONE;
      RW_STATUS.WRITE(SIMULATOR_STATUS);
      delay 2.0;                    -- wait for rest of tasks to startup
      -- main tasking simulated loop
 loop
      RW_COMMAND.READ (SIMULATOR_COMMAND);
      RW_PARAMETER.READ(SIMULATOR_PARAMETER);
      if ( SIMULATOR_COMMAND.COMMAND_NO /= LAST_COMMAND_NO )
      then
            -- simulate executing command - with wait
      SIMULATOR_STATUS.ECHO_COMMAND_NO :=
            SIMULATOR_STATUS.ECHO_COMMAND_NO + 1 ;
      SIMULATOR_STATUS.STATUS_MODE := EXECUTING;
      RW_STATUS.WRITE(SIMULATOR_STATUS);

                        -- parameter simulation
      RW_PARAMETER.WRITE(SIMULATOR_PARAMETER);

                        -- display simulation
      put("Simulator Command: ");
      enum_io.put(SIMULATOR_COMMAND.MODE);
      put(" Status Mode: ");
      enum_status_io.put(SIMULATOR_STATUS.STATUS_MODE);
      put(" Command # ");
      sim_int_io.put(SIMULATOR_COMMAND.COMMAND_NO);
      new_line;
      delay 1.0;
                        -- simulate command completed in status
      LAST_COMMAND_NO := SIMULATOR_COMMAND.COMMAND_NO ;
      SIMULATOR_STATUS.STATUS_MODE := DONE;
      RW_STATUS.WRITE(SIMULATOR_STATUS);
end if;

if( SIMULATOR_COMMAND.MODE = MARCHING) then
            -- parameter simulation
      simulator_parameter.x := simulator_parameter.x + 1;
      simulator_parameter.y := simulator_parameter.y + 2;
      simulator_parameter.z := simulator_parameter.z + 3;
      simulator_parameter.j(5) := simulator_parameter.j(5)
                        + 0.1 * simulator_command.energy;
      RW_PARAMETER.WRITE(SIMULATOR_PARAMETER);
            -- march to speed given
      delay SIMULATOR_COMMAND.SPEED;
 end if;
 delay 0.2;-- allow other processor chance
end loop;

 exception
      when others =>
            put_line("SIMULATOR TASK TERMINATED");
            raise;
end SIMULATOR_SYSTEM;
end SAMPLE_SIMULATOR;
```

```
File: oi.a - Main DDD Definitions and Run-time Executive

-------------------------------------------------------------
-- Section 1. with ADA compiler packages. Need system
-- package for address definitions.
-------------------------------------------------------------
with TEXT_IO; use TEXT_IO;
with system;

-------------------------------------------------------------
-- Section 2. Operator Interface package inclusion
-------------------------------------------------------------
with BUFFER_ADDRESSES     use BUFFER_ADDRESSES;
with OI_COMMAND_LINE;     use OI_COMMAND_LINE;
with OI_GLOBALS;          use OI_GLOBALS;
with OI_X;                use OI_X;
with OI_RCS;
-------------------------------------------------------------
-- Section 3. User Supplied Data Definition & simulator packages
-------------------------------------------------------------
with DATA_DEFINITIONS;    use DATA_DEFINITIONS;
with SAMPLE_SIMULATOR;    use SAMPLE_SIMULATOR;


procedure OPERATOR_INTERFACE is

-------------------------------------------------------------
-- Section 4. Generic nop callback. Always include.
-------------------------------------------------------------
      procedure callback renames oi_x.callback;

-------------------------------------------------------------
-- Section 5. Instantiate new buffer_addresses file reading procedure with types defined
-- in sample spec. Thus, ELEMENTS uses the BUFFER_NAMES enumerated
-- type. When invoked, the routine will read the buffer_addresses.dat file.
-------------------------------------------------------------
      procedure SAMPLE_BUFFER_INIT is new BUFFER_ADDRESSES.INIT(ELE-
MENTS => BUFFER_NAMES);

-------------------------------------------------------------
-- Section 6. Data routing definitions to visual interface
-- rw address is supplied at run-time
-------------------------------------------------------------

package OI_SAMPLE_COMMAND_ECHO is new OI_X.MENU_BUILD
      (
      NAME_IN                 => "Command Selection",
      NAME_OUT                => "Command List",
      VAR_ADDR                => LOCAL_COMMAND.MODE'ADDRESS,
      ENUM                    => PROCESSING_MODES,
      UPDATE                  => 10,
      RW_ENTRY                => TRUE,
      RW_NAME                 => "RW_COMMAND",
      RW_SIZE                 => COMMAND_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR      => LOCAL_COMMAND'ADDRESS,
      FILE_NAME               => "command_trace.dat",
      ENTRY_USE               => OI_X.SHOW
      );

package OI_SAMPLE_COMMAND_NO_ECHO is new OI_X.NUMERIC_1D_RCS_-
TO_USER_BUILD
      (
      NAME                    => "Echo Command No : ",
      VAR_ADDR                => LOCAL_STATUS.ECHO_COMMAND_-
NO'ADDRESS,
      DATA_TYPE               => OI_X.LONG,
      UPDATE                  => 10 ,
      RW_ENTRY                => TRUE,
      RW_NAME                 => "RW_STATUS",
      RW_SIZE                 => STATUS_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR      => LOCAL_STATUS'ADDRESS,
      FILE_NAME               => "cmd_no.dat",
      ENTRY_USE               => OI_X.SHOW
      );

package OI_SAMPLE_XZ_READ is new OI_X.NUMERIC_2D_USER_TO_RCS_BUILD
      (
      NAME                    => "XZ Coordinate",
      X_VAR_ADDR              => LOCAL_PARAMETER.X'ADDRESS,
      Y_VAR_ADDR              => LOCAL_PARAMETER.Z'ADDRESS,
      DATA_TYPE               => OI_X.LONG,
      UPDATE                  => 10 ,
      RW_ENTRY                => TRUE,
      RW_NAME                 => "RW_PARAMETER",
      RW_SIZE                 => PARAMETER_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR      => LOCAL_PARAMETER'ADDRESS,
      CALLBACK                => DATA_DEFINITIONS.SAMPLECALLBACK
      );

package OI_SAMPLE_STATUS_ECHO is new OI_X.ENUM_RCS_TO_USER_BUILD
      (
      NAME                    => "Sample Status : ",
      VAR_ADDR                => LOCAL_STATUS.STATUS_MODE'ADDRESS,
      UPDATE                  => 10 ,
      RW_ENTRY                => TRUE,
      RW_NAME                 => "RW_STATUS",
      RW_SIZE                 => STATUS_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR      => LOCAL_STATUS'ADDRESS,
      ENUM                    => STATUS_MODES
      );

package OI_SAMPLE_SPEED is new OI_X.NUMERIC_1D_USER_TO_RCS_BUILD
      (
      NAME                    => "Marching Speed ",
      VAR_ADDR                => LOCAL_COMMAND.SPEED'ADDRESS,
      DATA_TYPE               => OI_X.TIMED,
      UPDATE                  => 10 ,
      RW_ENTRY                => TRUE,
      RW_NAME                 => "RW_COMMAND",
      RW_SIZE                 => COMMAND_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR => LOCAL_COMMAND'ADDRESS
      );

package OI_SAMPLE_TIMING_BOUNDS is new
            OI_X.NUMERIC_ND_RCS_TO_USER_BUILD
      (
      NAME                    => "OI Speed Timing Bounds : ",
      ARRAY_ADDR              => TIMING_BOUNDS'ADDRESS,
      DATA_TYPE               => OI_X.DOUBLE,
      NUM_IN_ARRAY            => 2,
      UPDATE                  => 10 ,
      RW_ENTRY                => FALSE -- no need to use rw
      );

package OI_SAMPLE_JOINT_5 is new OI_X.NUMERIC_1D_RCS_TO_USER_BUILD
      (
      NAME                    => "Joint 5 : ",
      VAR_ADDR                => LOCAL_PARAMETERJ(5)'ADDRESS,
      DATA_TYPE               => OI_X.SHORT_FLOATING,
      UPDATE                  => 10 ,
      RW_ENTRY                => TRUE,
      RW_NAME                 => "RW_PARAMETER",
      RW_SIZE                 => PARAMETER_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR => LOCAL_PARAMETER'ADDRESS,
      FILE_NAME               => "joint5.dat",
      ENTRY_USE               => OI_X.SHOW
      );

package OI_SAMPLE_JOINT_ARRAY is new
       OI_X.NUMERIC_ND_RCS_TO_USER_BUILD
      (
      NAME                    => "Joint Array : ",
      ARRAY_ADDR              => LOCAL_PARAMETERJ(1)'ADDRESS,
      DATA_TYPE               => OI_X.SHORT_FLOATING,
```

```
      NUM_IN_ARRAY              => 7,
      UPDATE                    => 10,
      RW_ENTRY                  => TRUE,
      RW_NAME                   => "RW_PARAMETER",
      RW_SIZE                   => PARAMETER_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR        => LOCAL_PARAMETER'ADDRESS,
      FILE_NAME                 => "joint.dat",
--    ENTRY_USE                 => OI_X.BOTH -- if recording up invoking
      ENTRY_USE                 => OI_X.SHOW
      );

package OI_SAMPLE_COMMAND_INCR is new OI_X.CMD_INCR_BUILD
      (
      NAME                      => "Incr Sample Command No",
      VAR_ADDR                  => COMMAND_NO'ADDRESS,
      RW_ENTRY                  => TRUE,
      RW_NAME                   => "RW_COMMAND",
      RW_SIZE                   => COMMAND_TYPE'SIZE,
      RW_LOC_BUFFER_ADDR        => LOCAL_COMMAND'ADDRESS
      );
Lxs
begin
                        -- parse the command line run time parameters
  OI_COMMAND_LINE.PARSE;
                        -- initialize rw buffers memory locations
  SAMPLE_BUFFER_INIT;
                        -- initialize opint command buffers, etc.,
  DATA_DEFINITIONS.INIT;
                        -- user initializes local buffers before sending
                        -- out image to X
  OI_SAMPLE_COMMAND_ECHO.INIT(BUFFER_ADDR(to_int(RW_COMMAND_BUFFER)));
  OI_SAMPLE_SPEED.INIT(BUFFER_ADDR(to_int(RW_COMMAND_BUFFER)));
  OI_SAMPLE_TIMING_BOUNDS.INIT(BUFFER_ADDR(to_int(RW_COMMAND_BUFFER)));
  OI_SAMPLE_STATUS_ECHO.INIT(BUFFER_ADDR(to_int(RW_STATUS_BUFFER)));

  OI_SAMPLE_XZ_READ.INIT(BUFFER_ADDR(to_int(RW_PARAMETER_BUFFER)));
  OI_SAMPLE_JOINT_5.INIT(BUFFER_ADDR(to_int(RW_PARAMETER_BUFFER)));
  OI_SAMPLE_JOINT_ARRAY.INIT(BUFFER_ADDR(to_int(RW_PARAMETER_BUFFER)));
  OI_SAMPLE_COMMAND_INCR.INIT(BUFFER_ADDR(to_int(RW_COMMAND_BUFFER)));
                        -- initialize oi system, no tasking before this proc.
  OI_RCS.INIT;
  if(SIMULATOR)             -- start up simulator with signal
     then SAMPLE_SIMULATOR.SIMULATOR_SYSTEM.startup;
  end if;
                        -- startup up input, process,out
                        -- put tasks - no return from tasking
                        -- doesn't return until quit from window mgr.
  OI_RCS.PROCESS;
                        -- specialized program termination
  if(simulator) then
     abort SAMPLE_SIMULATOR.SIMULATOR_SYSTEM ;
  end if;

 exception
 when others =>
      put_line("UH OH! MAIN OI PROCEDURE ABNORMALLY TERMINATED");
end OPERATOR_INTERFACE;
```

Appendix D: Data Dictionary Definition - Generic Parameter Descriptions

```
Generic DDD Parameter Declarations
  -- 1-D NUMERIC X/USER TO RCS ENTRY BUILD : 1 number out at a time
       NAME                  : in string;                    -- name of the data routing entry
                                                             -- Reader/writer specs
       VAR_ADDR              : in system.address;            -- actual data parameter address to be routed
       DATA_TYPE             : NUMERIC_TYPES := LONG;        -- numeric representation
       UPDATE                : INTEGER:=0;                   -- update rate
                                                             -- numeric attributes
                                                             -- Reader/writer linkages
       RW_ENTRY              : boolean :=FALSE;              -- reader/writer entry? assume not.
       RW_NAME               : string := "NO_RW";            -- name of the reader/writer
       RW_SIZE               : integer := 0;                 -- size of reader/writer buffer in bits
       RW_LOC_BUFFER_ADDR    : in system.address:=           -- local reader/writer buffer address
                               system.address'ref(0);
       FILE_NAME             : string := "NONE";             -- default data logging file name to save values
       ENTRY_USE             : ENTRY_USE_types := SHOW;      -- type of data routing
       LOGGING_TYPE          : LOGGING_TYPES := TEXT;        -- initial type of data logging

  -- 1-D NUMERIC RCS TO X-USER ENTRY BUILD : 1 number out at a time
       NAME                  : in string;                    -- name of the data routing entry
       VAR_ADDR              : in system.address;            -- actual data parameter address
       DATA_TYPE             : NUMERIC_TYPES := LONG;        -- numeric representation
       UPDATE                : INTEGER:=0;                   -- update rate
                                                             -- Reader/writer linkages
       RW_ENTRY              : boolean :=FALSE;              -- reader/writer entry? assume not.
       RW_NAME               : string := "NO_RW";            -- name of the reader/writer
       RW_SIZE               : integer := 0;                 -- size of reader/writer buffer in bits
       RW_LOC_BUFFER_ADDR    : in system.address:=           -- local reader/writer buffer address
                               system.address'ref(0);
       FILE_NAME             : string := "NONE";             -- default data logging file name to save values
       ENTRY_USE             : ENTRY_USE_types := SHOW;      -- type of data routing
       LOGGING_TYPE          : LOGGING_TYPES := TEXT;        -- initial type of data logging
       with procedure CALLBACK is <>                         -- callback to routine for user mods

  -- N-D NUMERIC RCS TO X-USER ENTRY BUILD : n numbers out at a time
       NAME                  : in string;                    -- name of the data routing entry
                                                             -- numeric attributes
       ARRAY_ADDR            : in system.address;            -- address of array to be routed
       DATA_TYPE             : NUMERIC_TYPES := LONG;        -- numeric representation
       NUM_IN_ARRAY          : integer;                      -- number or elements in vector array
       GAP_SIZE              : integer := 0;                 -- size of record between entries
                                                             -- if zero, use data type for calc.
       UPDATE                : INTEGER:=0;                   -- update rate
                                                             -- Reader/writer spec.
       RW_ENTRY              : boolean :=FALSE;              -- reader/writer entry? assume not.
       RW_NAME               : string := "NO_RW";            -- name of the reader/writer
       RW_SIZE               : integer := 0;                 -- size of reader/writer buffer in bits
       RW_LOC_BUFFER_ADDR    : in system.address:=           -- local reader/writer buffer address
                               system.address'ref(0);
       FILE_NAME             : string := "NONE";             -- default data logging file name to save values
       ENTRY_USE             : ENTRY_USE_types := SHOW;      -- type of data routing
       LOGGING_TYPE          : LOGGING_TYPES := TEXT;        -- initial type of data logging
       with procedure CALLBACK is <>;                        -- callback to routine for user mods

  -- 2-D NUMERIC INPUT ENTRY BUILD
       NAME                  : in string;                    -- name of data routing entry
       X_VAR_ADDR            : in system.address;            -- x value address
       Y_VAR_ADDR            : in system.address;            -- y value address
       UPDATE                : INTEGER:=0;                   -- update rate
                                                             -- Numeric attributes
       DATA_TYPE             : NUMERIC_TYPES := LONG;        -- numeric representation
                                                             -- Reader/writer specification
       RW_ENTRY              : boolean :=FALSE;              -- reader/writer entry? assume not.
       RW_NAME               : string := "NO_RW";            -- name of the reader/writer
       RW_SIZE               : integer := 0;                 -- size of reader/writer buffer in bits
       RW_LOC_BUFFER_ADDR    : in system.address:=           -- local reader/writer buffer address
                               system.address'ref(0);
                                                             -- Data routing parameters
       FILE_NAME             : string := "NONE";             -- default data logging file name to save values
       ENTRY_USE             : ENTRY_USE_types := SHOW;      -- type of data routing
```

```
    LOGGING_TYPE            : LOGGING_TYPES := TEXT;        -- initial type of data logging
    with procedure CALLBACK is <>;                          -- callback to routine for user mods

-- ENUMERATED INPUT/OUTPUT ENTRY BUILD : vis a vis MENU
    NAME_IN                 : in string;                    -- name of menu output (to user) list
    NAME_OUT                : in string;                    -- name of menu selection (from user)
    VAR_ADDR                : in system.address;            -- location of enumerated type parameter
    type ENUM               is (<>);                        -- enumerated list type
    UPDATE                  : INTEGER:=0;                   -- update rate
                                                            -- Reader/writer specification
    RW_ENTRY                : boolean :=FALSE;              -- reader/writer entry? assume not.
    RW_NAME                 : string := "NO_RW";            -- name of the reader/writer
    RW_SIZE                 : integer := 0;                 -- size of reader/writer buffer in bits
    RW_LOC_BUFFER_ADDR      : in system.address:=           -- local reader/writer buffer address
                              system.address'ref(0);
                                                            -- Data routing parameters
    FILE_NAME               : string := "NONE";             -- default data logging file name to save selections
    ENTRY_USE               : ENTRY_USE_types := SHOW;      -- type of data routing
    LOGGING_TYPE            : LOGGING_TYPES := TEXT;        -- initial type of data logging
    with procedure CALLBACK is <>;                          -- callback to routine for user mods

-- ENUMERATED OUTPUT ENTRY BUILD
    NAME                    : in string;                    -- name of the data routing entry
    VAR_ADDR                : in system.address;            -- location of enumerated type parameter
    type ENUM               is (<>);                        -- enumerated list type
    UPDATE                  : INTEGER:=0;                   -- update rate
                                                            -- Reader/writer specs
    RW_ENTRY                : boolean :=FALSE;              -- reader/writer entry? assume not.
    RW_NAME                 : string := "NO_RW";            -- name of the reader/writer
    RW_SIZE                 : integer := 0;                 -- size of reader/writer buffer in bits
    RW_LOC_BUFFER_ADDR      : in system.address:=           -- local reader/writer buffer address
                              system.address'ref(0);
                                                            -- Data routing parameters
    FILE_NAME               : string := "NONE";             -- default data logging file name to save selections
    ENTRY_USE               : ENTRY_USE_types := SHOW;      -- type of data routing
    LOGGING_TYPE            : LOGGING_TYPES := TEXT;        -- initial type of data logging
    with procedure CALLBACK is <>;                          -- callback to routine for user mods

-- STRING INPUT ENTRY BUILD
    NAME                    : in string;                    -- name of the data routing entry
    VAR_ADDR                : in system.address;            -- location of enumerated type parameter
    UPDATE                  : INTEGER:=0;                   -- update rate
                                                            -- Reader/writer specs
    RW_ENTRY                : boolean :=FALSE;              -- reader/writer entry? assume not.
    RW_NAME                 : string := "NO_RW";            -- name of the reader/writer
    RW_SIZE                 : integer := 0;                 -- size of reader/writer buffer in bits
    RW_LOC_BUFFER_ADDR      : in system.address:=           -- local reader/writer buffer address
                              system.address'ref(0);
                                                            -- Data routing parameters
    FILE_NAME               : string := "NONE";             -- default data logging file name to save selections
    ENTRY_USE               : ENTRY_USE_types := SHOW;      -- type of data routing
    LOGGING_TYPE            : LOGGING_TYPES := TEXT;        -- initial type of data logging
    with procedure CALLBACK is <>;                          -- callback to routine for user mods

-- INCREMENT COMMAND OF RW ENTRY BUILD
    NAME                    : in string;                    -- Name of data routing entry
    VAR_ADDR                : in system.address;            -- variable address to increment
                                                            -- Reader/writer specification
    RW_ENTRY                : boolean := FALSE;             -- reader/writer entry? assume not.
    RW_NAME                 : string := "NO_RW";            -- name of the reader/writer
    RW_SIZE                 : integer := 0;                 -- size of reader/writer buffer in bits
    RW_LOC_BUFFER_ADDR      : in system.address:=           -- local reader/writer buffer address
                              system.address'ref(0);
    with procedure CALLBACK is <>; -- callback to routine for user mods
```

NIST-114A
(REV. 3-89)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4471

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
JANUARY 1991

4. TITLE AND SUBTITLE

The TROI (TeleRobotic Operator Interface) User's Guide

5. AUTHOR(S)

Barry A. Warsaw and John L. Michaloski

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

[ ] DOCUMENT DESCRIBES A COMPUTER PROGRAM; SF-185, FIPS SOFTWARE SUMMARY, IS ATTACHED.

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This document provides an introduction to the TeleRobotic Operator Interface (*TROI*) system and a user guide to TROI programming and operation. TROI provides a flexible, extensible, object-oriented interface to the NASREM robot control system (*RCS*). It consists of two major portions, the X-window system Graphical User Interface (*GUI*), and the RCS's data-server interface modules. TROI provides a highly dynamic environment for interacting with the RCS. The user is able to view and modify state variables of a running control system, and to edit, save, and load graphical interface configurations while connected to a running control system. In this way, the user can interactively perform diagnostics, switch diagnostic contexts by creating and destroying interactive objects, and reconfigure data flow networks, allowing control of RCS operations without the costs of switching opint operating modes. TROI merges a user-initiated window system event model and an independent data driven event model into a single event stream

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

graphical user-interface; man-machine; object-oriented; operator interface; real-time robot control system; user-interface management system; X Window System

13. AVAILABILITY

[X] UNLIMITED

[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).

[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.

[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
45

15. PRICE
A03

ELECTRONIC FORM