

The Sammi® Application Development Kit

This paper describes **Sammi®**, Kinesix Software's application development toolkit for creating and deploying networked software systems with real-time, interactive graphics.

INTRODUCTION.....	1
SAMMI ARCHITECTURE.....	2
CREATING A SAMMI APPLICATION.....	5
SAMMI COMPONENTS.....	12
SAMMI OBJECTS.....	19
SAMMI COMMANDS.....	25
MISCELLANEOUS SAMMI FEATURES.....	28
API / USER INTERFACE INTERACTION.....	30
API EVENTS.....	33
CUSTOMIZING SAMMI.....	35
OPTIONAL SOFTWARE.....	39
ABOUT KINESIX.....	40

INTRODUCTION

Sammi® (Standards-based, Advanced Man-Machine Interface) is a client/server software development toolkit for creating graphical, networked or embedded applications that are data, event, and command driven. It consists of a graphical editor for creating user interfaces; multiple executable programs that manage the user interfaces and network communications during runtime; libraries and tools for developing distributed applications that communicate with the Sammi runtime programs and interact with end-users; and libraries and tools for customizing and enhancing the graphical editor and runtime programs.

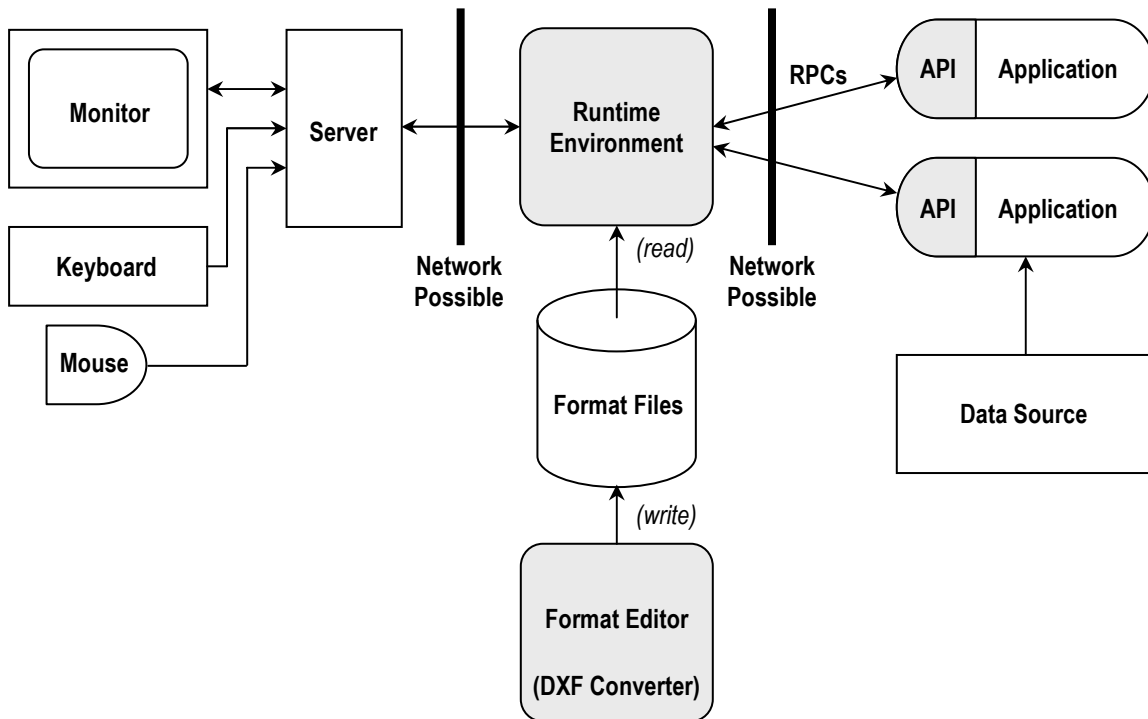
The Sammi development framework consists of two toolkits. The standard Sammi toolkit is the Application Development Kit (ADK), which is used to create user interfaces and the distributed application programs that communicate with the user interfaces through the Sammi runtime programs. The more comprehensive toolkit is the Sammi Development Kit (SDK), which is an object-oriented development system for customizing and enhancing the graphical editor and the Sammi runtime processes mentioned in the ADK.

The Sammi development framework is typically used to develop and deploy mission critical, real-time client/server command and control or process monitoring and control applications. Examples of these types of applications are found in air traffic control systems, satellite telemetry systems, electrical power distribution systems, nuclear reactor monitoring systems, and various resource distribution systems. However, since it contains a wide variety of general-purpose functionality, Sammi is not limited to these categories of applications and systems.

SAMMI ARCHITECTURE

In functional terms, Sammi consists of three components: the graphical editor, the Sammi runtime processes, and one or more server application programs. Figure 1 depicts how these components are organized in a Sammi application, and the following paragraphs contain high-level descriptions of the components. More detailed descriptions of the components are provided in later sections.

Figure 1: Application Diagram for a Typical UNIX Platform Deployment



The Sammi Graphical Editor

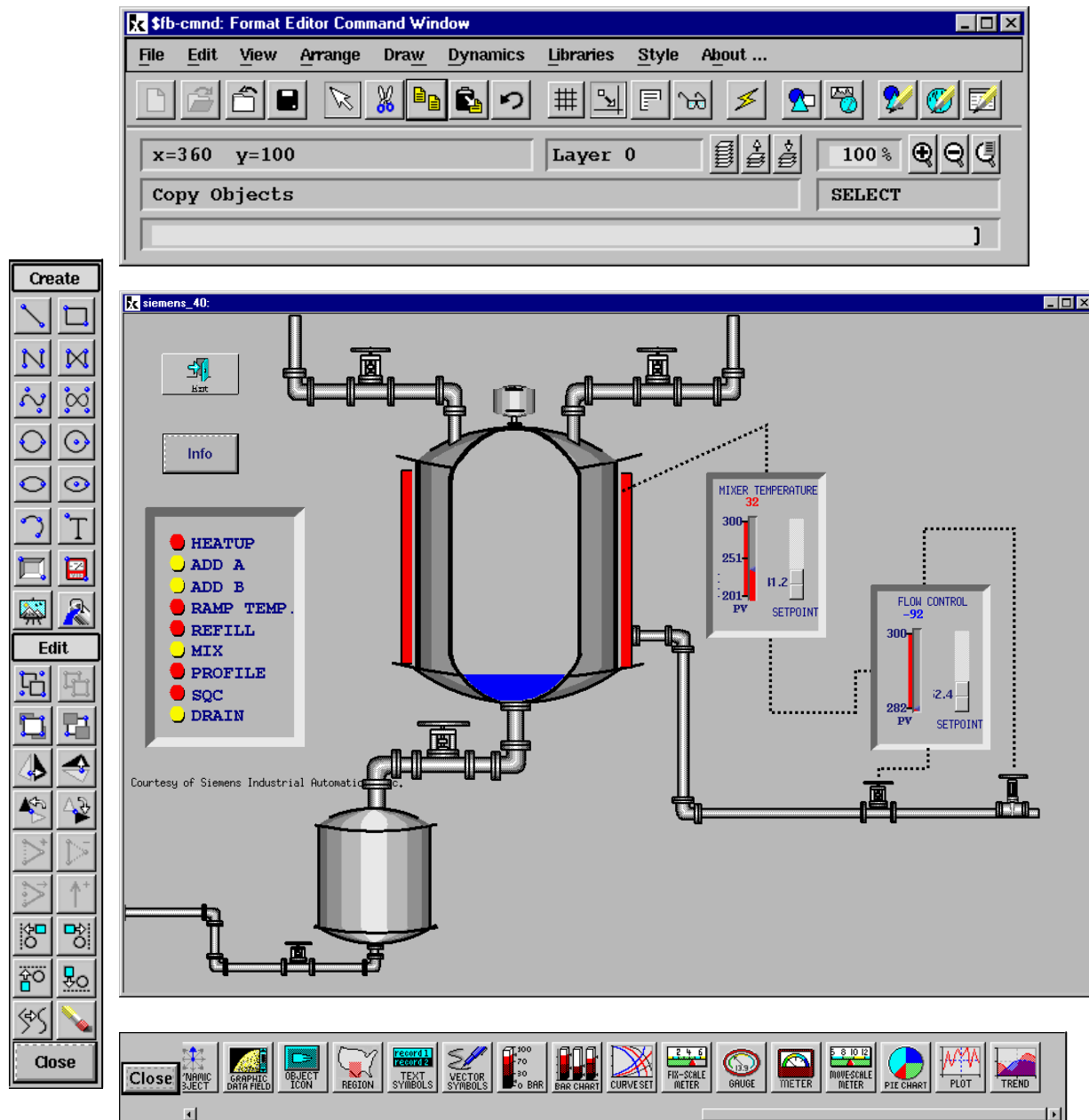
Sammi's graphical editor is called the Format Editor (FE). The FE combines the familiar look and feel of a PC drawing program with the functionality of a full-featured CAD program. It is used to graphically create the user interface components of a system and to establish the link between the user interface components and their controlling application processes.

The top-level user interface component created with the Format Editor is called a format. Each format is graphically displayed in a single window. Formats contain user interface components called Dynamic Display Objects (DDOs), and graphical primitives, called Static Display Objects (SDOs). DDOs are graphical objects that process commands and input events, display data, send and process commands, and interact with end users. They dynamically alter their graphical appearance based on events, commands, data values, and user interaction. Examples of DDOs are menus, meters, text fields, plots, and equations. SDOs are primitive graphical objects, such as lines, boxes and bitmap images. These objects, when they are first created, do not respond to user input, commands, or events, and their graphical appearances are fixed. These static objects can, however, be animated using a special DDO function, thus providing the ability to create a DDO out of any graphics primitive or composite group. A format can also contain a background image (such as a map) that forms a backdrop for the format's objects.

Formats, and all the objects they contain, are saved as binary images in loadable files called format files. The Sammi Runtime Environment loads the format files into shared memory, where they are accessible to the runtime processes.

The Format Editor's command, canvas, tool, and DDO palette windows are shown in Figure 2.

Figure 2: The Format Editor's Windows



The Sammi Runtime Environment

Sammi's Runtime Environment loads format files, creates windows for them, and activates the objects in them. It manages the interaction between Sammi users and DDOs, and manages network communications between Sammi's runtime processes and peer applications/data servers.

Multiple, cooperating processes constitute the Sammi Runtime Environment. Each process performs specific functions. For example, one process receives data and routes it to the correct formats and DDOs, another process causes display objects to blink, while another process handles input events. This distribution of tasks permits each Sammi process to perform its job as quickly and efficiently as it can.

Sammi's runtime processes communicate with each other and with server applications using remote procedure calls and shared memory. These runtime processes primarily exchange information with each other by reading from and writing to shared memory and communicate with peer applications and data servers through the Sammi Application Programming Interface, which internally uses Open Network Computing Remote Procedure Calls (ONC RPCs) to implement Sammi's network communications.

API Peer Applications and Data Servers

The peer-peer applications and data servers developed with the Sammi API implement the application-specific functionality of a Sammi software system. They are the executable programs that provide data and control logic for the overall system and for its user interface components. They send data and commands to the Sammi runtime processes; and receive data, commands, and API-defined event messages from the runtime processes. A single Sammi application can consist of multiple peer applications/data servers running on a single workstation, or running on multiple workstations distributed across a network.

The Sammi API defines the message data structures, event types, and protocol that peer applications/data servers and the Sammi runtime processes use to communicate with each other. The API consists of include files containing the message data structure, event, and procedure declarations; and a link library containing the procedures that implement the API protocol and network communications facilities. The API link library is implemented in C, and the procedures can be directly called from C or C++ code. In addition to the standard C bindings, optional ADA bindings for the API library are available.

Since Sammi user interfaces are developed entirely with the Format Editor and managed by the Sammi runtime processes, no embedded user interface code is required in API peer application/data server code. Additionally, since the Sammi API implements and manages network communications, no interprocess communications code is required in these applications/servers.

CREATING A SAMMI APPLICATION

A major percentage of the work involved in creating any software system with a feature rich graphical user interface is devoted to the design and layout of the interface and to the interaction between the user interface and the actual application code. For most modern software systems, at least seventy percent of the code is devoted entirely to the user interface. Creating an application with a graphical interface is difficult and time-consuming, but creating a networked software system with a graphical user interface is even more difficult. Resolving issues such as synchronization of multiple processes, differences in machine binary formats, application and network failures, and many other issues related to networked inter-process communications requires a great deal of time, effort, and expertise.

The Sammi application development framework separates an application's user interface components from its code and provides transparent network facilities for distributed applications. The user interface is created graphically with the Format Editor, and no code is required to create or modify the user interface. The Sammi Runtime Environment manages the user interface during runtime, and handles the user interaction with it. Thus, no low-level code to manage the interface during runtime is required. The API library encapsulates and manages both local and networked inter-process communications and data transfer, so Sammi application developers do not need to implement low-level network-related code.

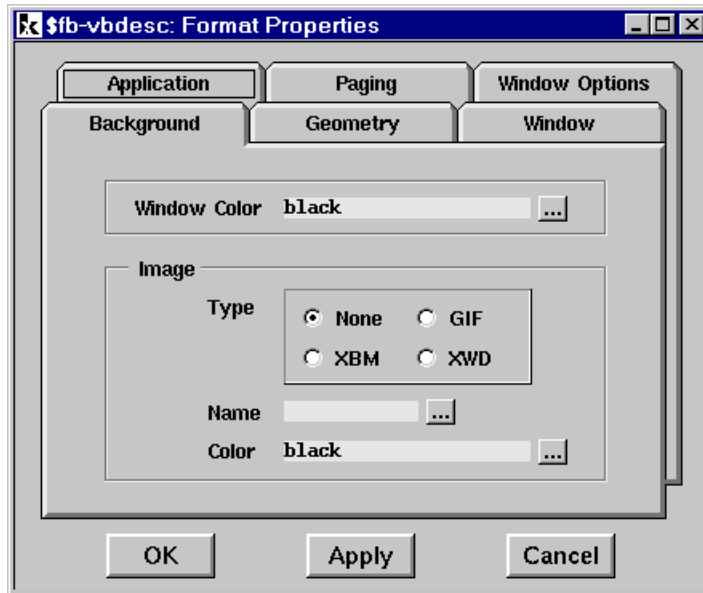
API applications and data servers created with the Sammi framework consist of three components: the graphical user interface, the linkage between the interface and the application code, and the API programs that drive and interact with the interface. The following paragraphs describe how these three components are created.

Creating the Graphical User Interface

All Sammi user interfaces consist of windows containing user interface components such as text entry fields, menus, etc. Windows are created graphically with the Format Editor, which involves creating the windows and setting up their attributes; adding background images, static graphics, and dynamic display objects to the windows; and linking the windows and their dynamic display objects to each other and to the API peer applications and data servers.

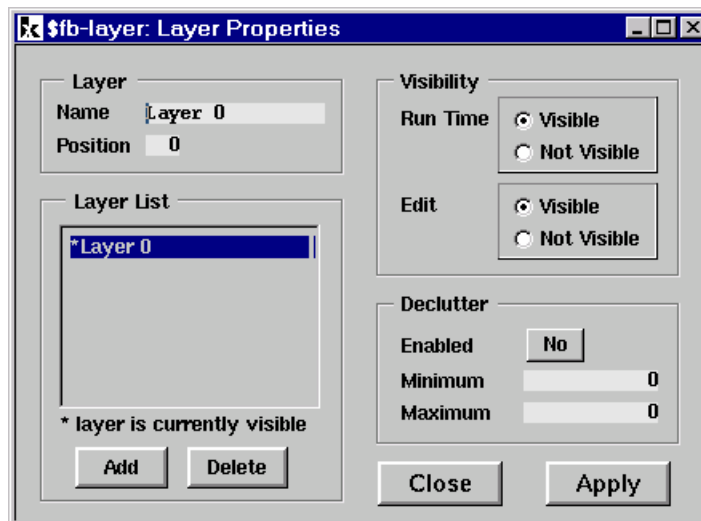
A window is created by simply selecting a "new format" icon or menu button. The Format Editor opens a canvas that corresponds to the window. The canvas is then sized and placed at the location where it is to appear when added to the screen by Sammi's runtime processes. The window's attributes are set by either loading them from a defaults file (with a "load defaults" menu selection), or manually through a property dialog. The dialog window is popped up with a "Format Properties" menu selection. The window's attributes can be set or changed at any time. The Format Property dialog window is shown in Figure 3.

Figure 3: The Format Property Dialog Window



The contents of Sammi windows consist of (optional) background images, layers, static display objects, and dynamic display objects. All static and dynamic display objects are contained in layers. Each window has at least one layer, which is created automatically when the window is created. Additional layers are added to a window by selecting a "layer" icon or menu item, then selecting an "add layer" button in the "layers" dialog that pops up. Layers can be added and deleted. Their visibility attribute (visible or invisible) can also be set for both the Runtime Environment and for the editing environment. Objects are added to the active edit layer, which is set using either the popup layers dialog, or with an icon on the Format Editor's command window. A window's attributes can be set or changed at any time. Figure 4 shows the Layer dialog window.

Figure 4: The Layer Dialog Window



Background images are added to windows with the format's property dialog. This requires entering the name of the file containing the image, and setting the image's foreground and background colors. The colors are selected from a popup color palette.

Static display objects are added to the window's active edit layer by selecting the type of object to add, then drawing the object on the canvas with the mouse. Object types are selected by selecting an icon from a "tool" dialog, or by selecting a menu item. Attributes for new objects (color, fill style, line width, etc) are set up with a popup attributes dialog or by loading them from a defaults file. After an object is created and placed, its attributes can be changed by simply double-clicking on the object, which pops up the attributes dialog, then entering the values. These values are entered by selecting attribute buttons or picking values from popup palettes or lists. An object's attributes can also be edited by single-click selecting the object, then picking one of several style menu items. The style menu items also pop up the *attributes* dialog. The object attribute edit dialog window is shown in the following figure (Figure 5):

Figure 5: The Object Attributes Dialog Window

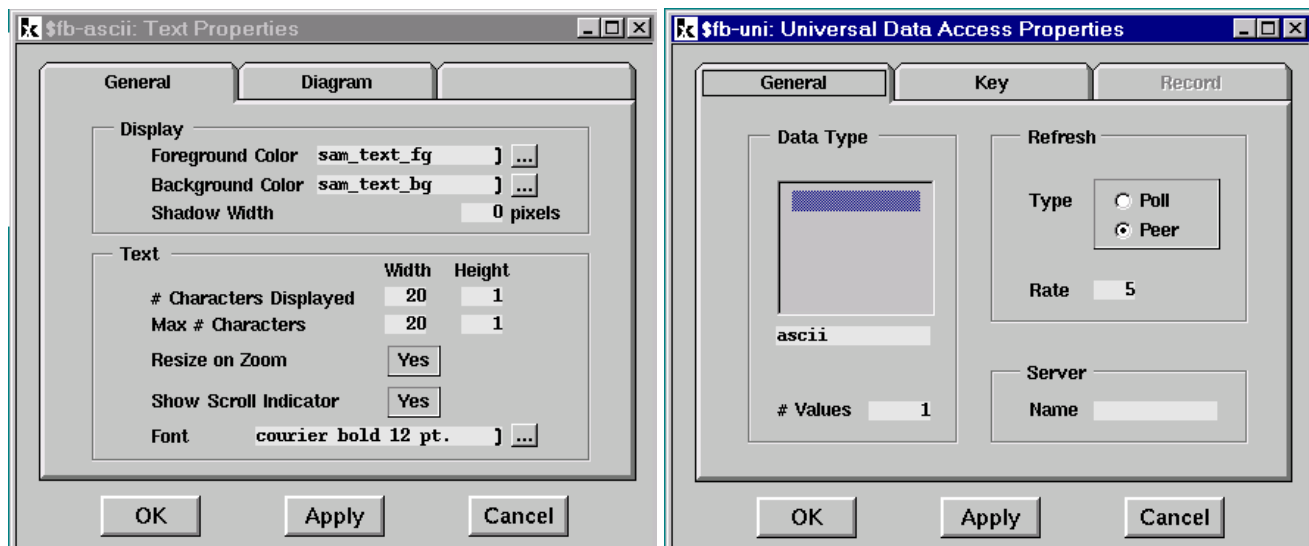


Dynamic display objects can be added to the window's active edit layer the same way static display objects are added: by selecting a "new DDO" icon from the tool dialog, then drawing the rectangle that defines the location and size of the DDO. When the DDO is placed and sized, the Format Editor pops up a DDO dialog window. This dialog is used to set up various attributes for the DDO, and to set its display type and data type. A DDO's display type is a sub-object that implements its graphical appearance and user interaction; i.e., a menu, a meter, a numeric input/output field, etc. The data type is a sub-object that manages the DDO's runtime data. These sub-objects and the architecture of a DDO are described in more detail in a later section. Both the display type and data type are set by selecting a sub object type from a select list. There are approximately forty-five built-in display types and five built-in data types that are used to create a DDO. When these types are selected, the Format Editor pops up dialog windows containing attributes for the sub objects. For example, the dialog window for a text field contains entries for its font, colors, shadow thickness, etc; and the dialog window for Sammi's *universal* data type contains entries for the DDO's read/write keys, logical server name, runtime value data type (short, long, etc.), and other information about the DDO's runtime data. The Dynamic Field Definition (DFD), display type, and data type dialog windows for a text DDO whose data access type is "universal" are depicted in Figures 6 and 7:

Figure 6: The DDO Edit Dialog Window



Figure 7: The Display and Data Type Edit Dialog Windows for a Text DDO



Another, simpler way to add a DDO to a window is by selecting a pre-built, library DDO and adding it to the window at the desired location. Library DDO's are created by creating a new DDO (as described above), setting up default values for the DDO, and saving it as a library part. Library DDOs are added to a window by selecting an icon representing the DDO from a DDO palette window, or selecting it from a menu, then simply dragging and dropping the icon in the window. DDOs are edited by double-click selecting them, which pops up the DDO dialog window. The display type and data type attributes are edited by selecting the display type and data type buttons on the DDO dialog window, which pops up the dialog window for the specified sub object.

In addition to library DDOs, the Format Editor supports two other types of library parts: *composites* and *reference objects*. Both composite library parts and reference objects are groups of objects that are saved as single parts. Both are added to a format by selecting their menu items or icons, entering the part name in a pop up dialog window, and dragging and dropping them into the format. There are two differences between composite and reference object library parts: composite parts are copied into a format and may contain DDOs; and reference object parts may not contain DDOs and are added to the format by reference instead of as a copy. Only one copy of a reference object exists in a global memory area that is accessible to referencing instances. The single copy is loaded when the Format Editor or Runtime Environment is started, and can be reloaded at any time. It is this single copy of a reference object that is accessed by its referencing instance objects. These two library parts are discussed in more detail in a later section.

After all the desired objects are placed in the window and their sizes and attributes are set, the window and its objects are saved by selecting a "save format" icon or menu item and entering the format's name in a popup dialog window. The Format Editor traverses the object hierarchy and sends a message to each object to write itself into a binary buffer. After the objects complete writing themselves into the binary buffer, the buffer is written to a format file. Except for encapsulated X Toolkit widgets (which are dynamically created at runtime), all the objects in a format file are completely constructed, their attributes are set, and space for their runtime data values is pre-allocated when the format is saved. The format can thus be loaded directly into memory and the objects can immediately begin performing their functions without incurring object creation and initialization overhead in the Runtime Environment (encapsulated X Toolkit widgets are the exception).

The Format Editor supports two modes of execution: prototyping and standard. The only differences between these two modes of execution are that in prototyping mode, all the Sammi runtime processes are also executing (including API applications and servers), and all the runtime facilities are accessible in the Format Editor. This is the mode that is used most often, and its advantage is that formats can be added to the screen and tested with live data, commands, and events as soon as they are saved in a format file. Application programmers can use the Format Editor in this mode to rapidly create, test, and modify formats without leaving the editor environment.

Linking the API Application to its User Interface

API applications are linked to their Sammi user interface components with logical server names, read/write keys, format and DDO identifiers, and command strings. Server names, read/write keys, and command strings are placed in DDOs in the Format Editor, either at the time the DDOs are created, or by editing them at a later time. Some Sammi DDOs also support dynamic runtime command definition by API applications. The command strings are part of the runtime data for these DDOs. Runtime keys can also be used to dynamically set or change server names and read/write keys during runtime.

API applications do not directly access or manage their user interface components at an implementation level. Instead, they maintain and manage them at an abstract, software model level using information from their application domains. DDO read/write keys identify the models, or parts of the models, in application domain terms. The DDOs that contain the read/write keys constitute the graphical view of the model at the presentation level.

For example, a process monitoring application may be concerned with the temperature and pressure inside a mixing chamber, and need to graphically present these values to a user in a variety of ways. In this domain, temperature and pressure values are elements of a software model of an actual mixing chamber, and could be identified with the read keys "chamber1.temperature" and "chamber1.pressure" (actual keys would probably not be this verbose). These read keys can be used with many different Sammi DDOs to present the values to an end user: a meter, a gauge, a simple numeric field, a trend, et al. In fact, different views of the values can be presented simultaneously by using the same read key in different DDOs. Any of these DDOs can be replaced with any other one without affecting the application's view of the model or how it interacts with the DDOs.

As another example, a command and control system for a space station is concerned with the status of the inner and outer doors of airlocks, and provides controls for users to open the doors, depending on the state of other elements of the airlocks. In this domain, each door is an element of a software model of an airlock, and could be identified with the read keys "airlock1.inner" and "airlock1.outer". These model elements could be represented at the presentation level with labeled TextTable DDOs that display the text strings "Open", "Opening", "Closing", and "Closed", based on the state of the corresponding door. The doors could also be represented with DDOs that move drawings of the doors as they open and close. Users open or close the doors with switches, which could be presented to the user graphically as toggle button DDOs that initiate a series of application program actions when toggled. The toggle buttons could be identified with the read keys "airlock1.innerswitch" and "airlock2.outerswitch". Since buttons are command objects,

their actions could be initiated with "open" and "close" commands that are sent to the application when the buttons are toggled.

As illustrated by the previous examples, the task of establishing the read/write keys to identify user interface components in an API application essentially occurs at the analysis and design stage of software development. The same is true for defining the commands an application will receive during runtime, and for defining logical server names. API application developers analyze their problem domain to determine the entities, their behavior, and the relationships between them to decide what needs to be modeled in software and how to partition the software into different applications. The partitioning results in one or more executable programs, which are then assigned logical server names. The entities (or their components) that are modeled in software are assigned read/write keys that identify the software structures or data elements used to model them. The behavior of the objects are modeled using code, commands, and/or commands that trigger the code. The server names, read/write keys, and commands are then added to the formats and DDOs that constitute the application's user-oriented presentation level; i.e., its graphical user interface.

Implementing the API Application

API-based applications and/or data servers provide the overall control and runtime data for their Sammi user interfaces. They do this by sending commands to the Sammi Runtime Environment and to each other, processing commands from user interface components, and by reading and writing data values from and to user interface components. The API provides the communications layer and implements the event protocol that makes this distributed interprocess interaction possible.

At the top level, all API applications and data servers are implemented using the same programming model and program structure. In fact, the program structure is so standard, a generic, "skeleton" program is provided with the Sammi ADK. API developers can simply flesh out the skeleton to create their executable program. To implement a complete API application, API developers need to decide whether their program needs to communicate with other API applications, which API event messages their application needs to receive, whether their program needs to send updates synchronously or asynchronously, what data structures are needed, and how the application's data are internally stored and retrieved. The following paragraphs discuss aspects of the API that are relevant to these considerations.

API-based applications are implemented using an event-driven programming model. The program performs initialization and setup, then enters an event loop. When an input event occurs, the program reads and processes the event message, then waits for another event. This wait-read-process loop continues until some event occurs which ends the program.

There are two techniques API applications can use to implement the event control logic. The first technique uses an explicit loop that reads an event message, examines the event type code to determine what kind of event occurred, then calls an application procedure to handle it. The second technique uses event callback procedures. With this technique, the application registers event callback procedures with the API, then calls the API event input procedure. When an event occurs and an event callback procedure is registered for the event, the API event input procedure calls the callback procedure to take care of it.

Normally, the API event input procedure blocks until an event message is available for reading. API applications can specify a timeout interval for the input procedure. If an event is not available within the specified timeout interval, the input procedure returns.

The Sammi API defines fourteen events, defines the data structures associated with the events, and specifies the protocol (what the event means and what the application is supposed to do) for the events. For example, one event is an "add window" event. Sammi sends this event to an application when a format containing one or more DDOs connected to the application is added to the screen. A data structure containing information about the DDOs is associated with this event, and the application is expected to return the runtime data for the DDOs using information in the event message's data structure. The API events are described in a later section.

API applications wait for events by calling an API procedure that reads and writes from sockets using ONC RPC procedure calls. The API library automatically creates the sockets required for the Runtime Environment. Applications can also open their own sockets and register a socket handler procedure that the API input procedure calls when the socket is ready for input or output. Instead of sockets, API applications can open RPC connections and register handler procedures for the connections with the API. API applications can thus communicate directly with

other API applications as well as with the Sammi Runtime Environment. As an alternative to opening sockets or RPC connections to communicate with other API applications, Sammi's built-in command dispatching facilities can be used to send commands between API applications.

In a typical Sammi Runtime Environment, a large number of data requests and data updates occur. To minimize the overhead associated with network transactions, both the API applications/servers and the Sammi processes buffer messages, then send all the buffered messages in one transaction instead of sending many small messages separately. The API provides the procedures and structures for buffering and flushing messages. API developers can control the number of messages that are buffered by flushing the buffered messages when necessary.

API applications normally send data updates synchronously; i.e., the RPC procedure that actually flushes the update messages to the Sammi runtime processes does not return until the updates are complete. As an alternative, API developers can choose to use asynchronous updates by calling an RPC procedure that queues the update messages and returns immediately. If asynchronous updates are used, the Sammi runtime processes notify the API application when they complete the queued updates.

How the data structures, data storage, and data retrieval mechanisms for an API application are implemented is independent of the API. Instead, these implementation details are dependent upon the problem being solved, the solution constraints, the solution environment, and the experience of the API developers. However, since the linkage between these internal constructs and the application's user interface components are primarily established with read/write keys and commands, they should be designed and implemented so they can be readily identified with keys and commands. For example, many API applications maintain internal tables and arrays containing state information and data values and use read/write keys to identify specific tables and table entries, or arrays and array entries. Using this technique, when Sammi's runtime processes request the runtime data for the application's DDOs, the application can immediately retrieve and return the values. Likewise, when a DDO's value is modified, the application can immediately evaluate the modification, update its internal values, and perform any needed processing.

SAMMI COMPONENTS

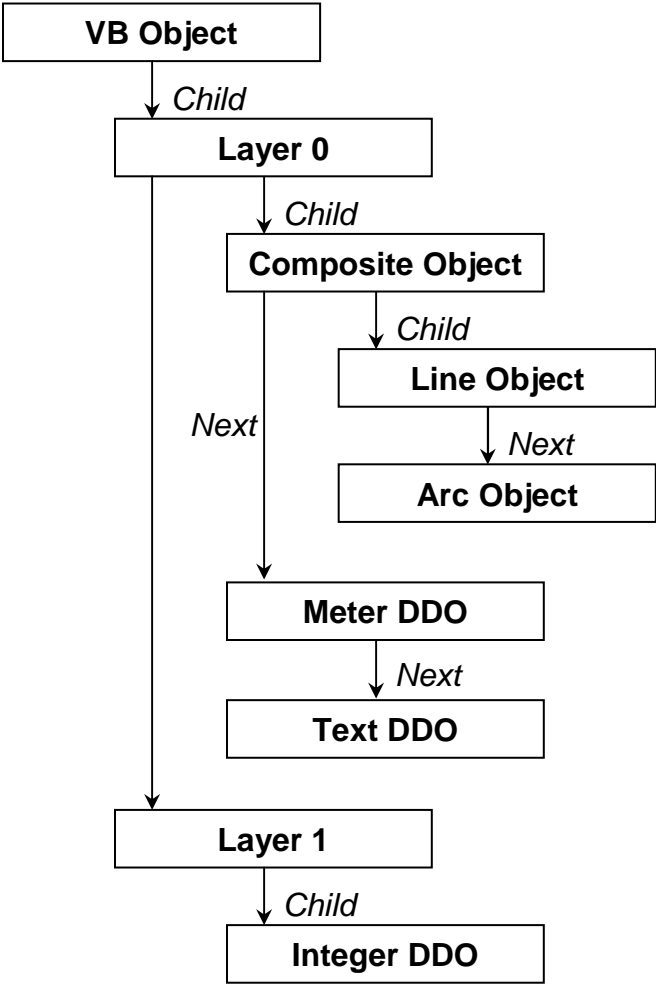
This section describes the major components of Sammi in detail.

Formats

A one-to-one correspondence exists between a Sammi format and a top-level shell window in runtime Sammi; each format displays itself and its child objects in a single window.

The objects in Sammi formats are organized in a tree data structure. This structure consists of parent and child objects. The top-level object in Sammi's object tree is called a Virtual Buffer (VB) object. Every Sammi format contains one VB object; it is the only object in a format that does not have a parent object. The child objects of a VB are layer objects, which are composite objects. They are used to organize groups of objects into planes, much like stacks of paper in a tray. The child objects of layers are either other composite objects, dynamic display objects, or static display objects. Figure 8 depicts the object hierarchy for a format containing two layers:

Figure 8: A Format's Object Hierarchy



Composite objects manage both themselves and their child objects. Because of this assignment of responsibility, internal control in Sammi processes is distributed, rather than centralized, and parallels the object hierarchy. For example, to display all the objects on a given layer, a "display" message is sent to the layer object. The layer object displays itself by iterating through its list of child objects, sending each one a message to display itself.

Dynamic Display Objects (DDOs)

DDOs are Sammi's fundamental user interface components. DDOs perform the following functions:

- DDOs receive and display data from external sources,
- DDOs receive commands and perform actions based on them,
- DDOs respond to input events, and
- DDOs send data, events, and commands to various external recipients.

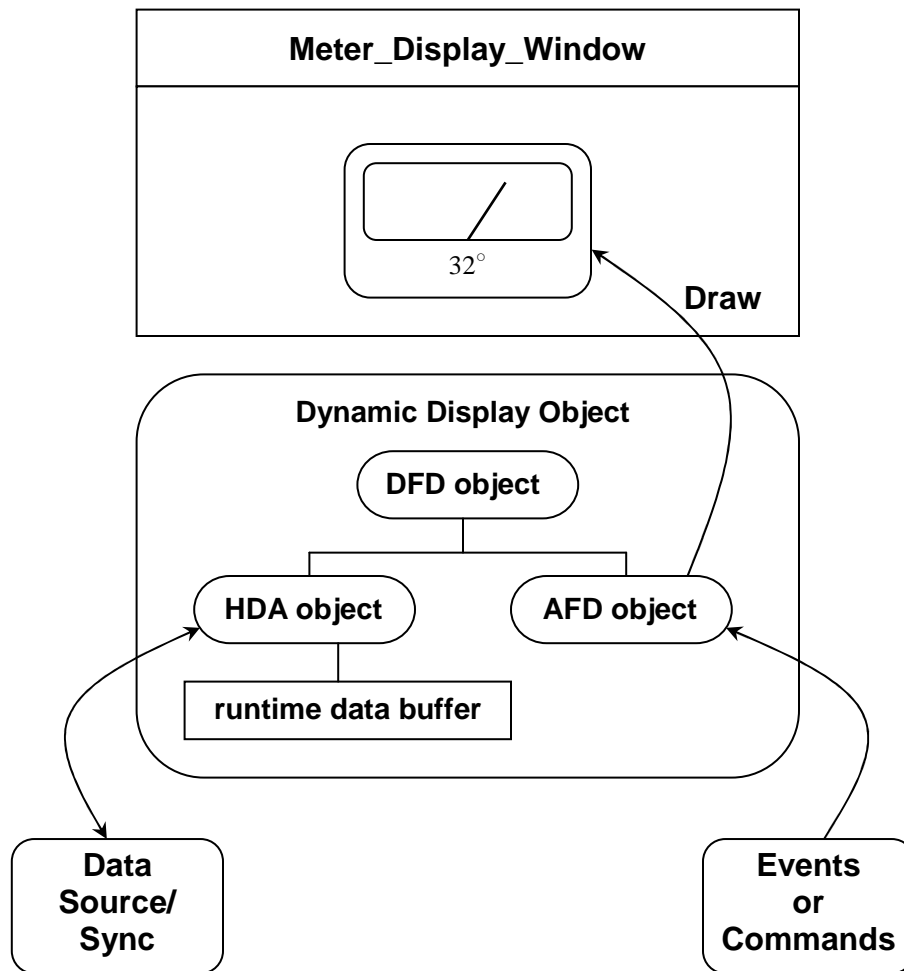
In a Sammi window, DDOs look and behave like single objects. They are actually complex objects composed of three separate objects. These three objects are Dynamic Field Definition (DFD) objects, Auxiliary Field Definition (AFD) objects, and Host Data Access (HDA) objects.

DFD objects are the principle objects in DDOs. They contain information that is common to a DDO in its entirety and that is shared by its DFD, AFD, and HDA objects. DFDs contain pointers to their DDO's AFD and HDA objects, and function as their controllers, sending and forwarding messages to them.

AFD objects are responsible for a DDO's "look and feel". They are the part of a DDO that is graphically visible in a window and with which users interact. AFDs are implemented with either graphics library calls and event handlers, or as containers for X Toolkit-based widgets (usually from the Motif widget set).

HDA objects are responsible for the connectivity between DDOs and their external data sources and syncs. They provide shared memory space for their DDO's runtime data and manage the movement of runtime data into and out of their DDOs. Figure 9 portrays the architecture of a DDO and the responsibilities of its three objects:

Figure 9: DDO Architecture



The Sammi Runtime Environment

The Sammi Runtime Environment loads and displays formats created in the Format Editor (or by a conversion program), manages interaction with end users, and manages the transfer of data and control information among Sammi processes and API peer applications/data servers. Three cooperating Sammi processes perform these functions. These processes share data and control information with each other using shared memory and two internal sockets. The following paragraphs describe these components of Sammi's Runtime Environment.

Shared Memory

In the Sammi runtime, data global to all runtime processes are placed in shared memory. Shared memory is allocated and global data placed into it by Sammi's initialization process when Sammi is started. Some of the global data, such as value tables, can be reloaded during runtime.

In addition to global data, binary format files are read directly into shared memory when formats are added to the screen. Thus, the data values and state variables for Sammi objects are available to and shared by all Sammi runtime processes.

Runtime data values for DDOs are written into and read from shared memory. To prevent collisions between processes writing to shared memory, each process obtains a shared memory lock before updating shared memory values, and releases the lock upon completion of the update action.

Formats are cached in shared memory. The cached formats are ordered using a least recently used (LRU) algorithm. The number of formats that are cached is specified in Sammi's configuration file. Formats remain in shared memory even when they are not displayed and are not removed until the space they occupy is needed for another format, unless they are marked as permanent formats.

Event Handler Process, `s2_event`

The event handler process is Sammi's main runtime process. It implements four major areas of runtime functionality:

- window management,
- event handling,
- widget management, and
- Sammi command processing.

As part of its event-handling functionality, `s2_event` notifies external API processes when a DDO's data values are modified through user input.

This is the only Sammi runtime program that is linked with the X Toolkit and with the Motif widget library. As a result, in a UNIX application, widgets for widget-based DDOs exist only in `s2_event`'s process space, although the DDOs that encapsulate the widgets exist in shared memory and are accessible in all Sammi runtime processes.

`s2_event` creates, caches, and destroys the Xt shell widgets that are the top-level windows for formats. It handles all input events for runtime Sammi using event-handler and widget callback procedures. These procedures manage all user interaction with formats and DDOs.

`s2_event` creates and manages a command socket and an expose socket. These sockets are used internally by the Sammi runtime processes. The command socket receives commands forwarded to `s2_event` from external API applications, along with commands issued by other Sammi processes.

The expose socket is used by widget-based DDOs to send update commands to their widgets from Sammi's stream data and dynamic refresh processors. `s2_event` also provides a local (non socket-based) facility that is used to process commands typed into Sammi's command window and to dispatch commands to and from internal procedures and DDOs.

Streamed Data Manager Process, `s2_stream`

The streamed data manager process is the Sammi runtime process that implements peer-to-peer data transfer from external API applications into runtime Sammi. `s2_stream` receives data values sent to DDOs by their peer API applications, places the received data values into shared memory, and then sends an update message to each DDO whose runtime data values were updated. DDOs read their runtime data values and redisplay their graphics and/or text strings when they receive an update message.

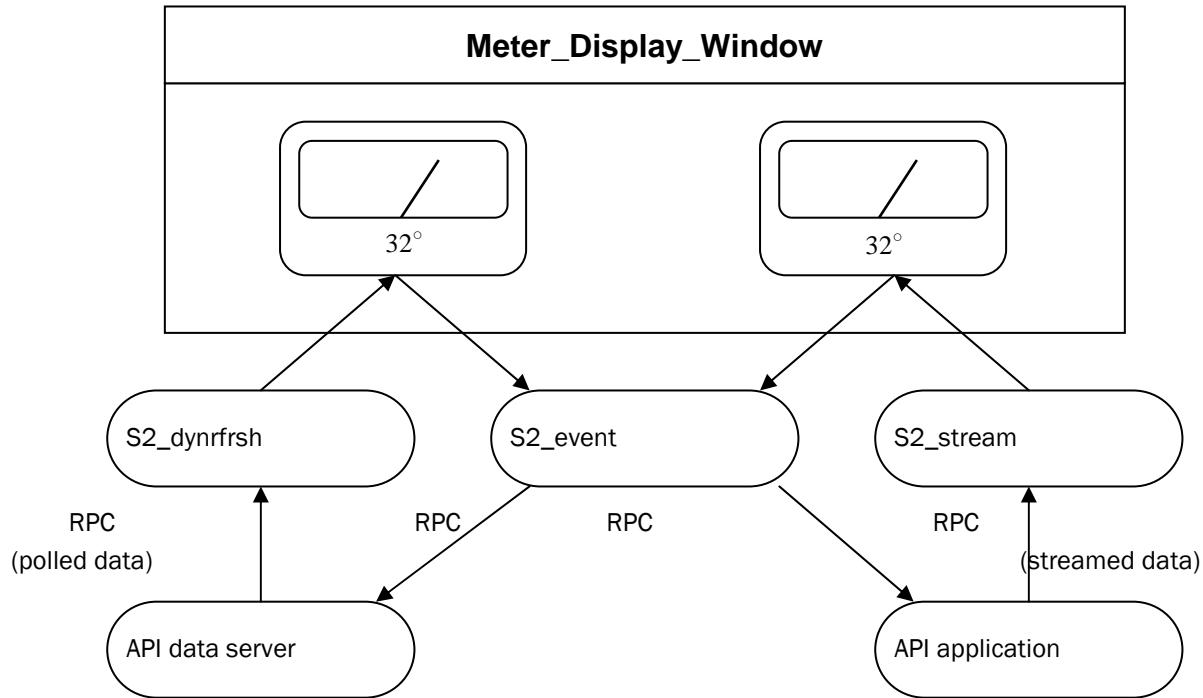
Polled Data Manager Process, `s2_dynrfrsh`

The polled data manager process implements periodic (polled) data transfer from API data servers. `s2_dynrfrsh` requests data for DDOs in a displayed format at periodic intervals (poll rate). The data request intervals are determined by the refresh rates specified for DDOs when they are created. `s2_dynrfrsh` receives and places the requested data values in shared memory, then sends update messages to the DDOs. API peers or data servers can send data updates to polled DDOs at any time. However, unlike `s2_stream`, which updates DDOs as soon as new data arrives for them, `s2_dynrfrsh` does not update the DDOs until their next refresh time occurs.

The following diagram (Figure 10) shows how runtime data flows between API applications, Sammi's runtime

processes, and Dynamic Display Objects:

Figure 10: Runtime Data Flow



Common Processes

Several processes are common to both the Format Editor and to the Runtime Environment. They are:

System Initialization Process, **s2_sysinit**

This is the first Sammi process that starts executing when the Format Editor and the Sammi runtime processes are run. It allocates Sammi's shared memory segment and loads global values into it. The size of the shared memory segment, as well as some of the values that are loaded into it, are read from a modifiable configuration file. In addition to values from the configuration file, **s2_sysinit** reads tables and values from editable files into shared memory. Some of these tables and values are bitmap, pixmap, and symbol tables; runtime annotation tables; and preferences values.

Command Forwarding Process, **s2_evtsvr**

This process receives commands and forwards them to Sammi's runtime event handler process or to the Format Editor. The commands are forwarded through Sammi's internal command socket. The command forwarding process and the internal command socket are used to prevent Sammi's runtime communications channels from becoming deadlocked with commands from peer-to-peer applications. Figure 11 illustrates how commands flow between **s2_evtsvr**, **s2_event**, and API applications:

Figure 11: Runtime Environment Command Flow

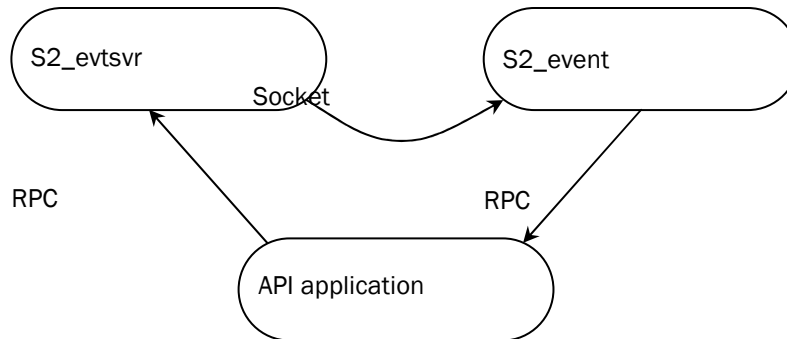


Table Reload Process, s2_contbl

This process reloads Sammi's global tables during runtime. Like the system initialization process, it reads values from editable files and constructs internal tables. However, instead of constructing the internal tables directly in Sammi's shared memory (which would require locking shared memory for an unacceptable amount of time), s2_contbl constructs them in its process space. After the tables are constructed, the table reload process locks shared memory, replaces the tables in shared memory, and releases the shared memory lock. In this way, shared memory is locked for only a brief period.

Color Blink Process, s2_blink

A major use of runtime annotations is to accentuate the graphical display of critical values. An effective way to do this is by "blinking" the displayed values. This is the function of the color blink process. It does its job by swapping color values in Sammi's color-map at periodic intervals. Using a separate process to perform this task eliminates redundant code from Sammi's core runtime processes and reduces redisplay time.

Error Logging Process, s2_error

This process creates and manages error log files for the Format Editor and the Sammi runtime processes. When an error condition is detected by a Sammi program, it makes a remote procedure call to this process, sending it a message describing the error, a severity level, the name of the procedure that detected the error, and a process identifier that identifies which Sammi process detected the error. The error logging process formats the error information, then writes it to an error file along with the time the error occurred. Sammi provides a utility program that formats and displays the contents of the error file.

Alarm Process, s2_alarm

A Sammi alarm is a packet of data a remote process sends to a Sammi runtime when a specified "trigger" event occurs. Typically, events that trigger alarms are critical values; i.e., values that exceed or fall below some specified limit. These events usually require the immediate attention of an end user, or may need to be logged in a special archival file. In addition to critical events, "special" Sammi events are captured and processed by Sammi's alarm process. These special events are events such as log on events and data entry events.

Sammi's alarm process implements event logging, alarm notification, alarm archival, and formatted printing of event and alarm files. Sammi's alarm functionality is actually implemented with several API-based peer applications and data filtering and formatting control files. However, s2_alarm is Sammi's main alarm process.

The alarm process provides a wide range of options for defining event filters, filtering events and alarms, displaying and highlighting them, and obtaining alarm acknowledgement from users.

Task Scheduler, s2_schedule

The task scheduler process is used to execute tasks in the Sammi Runtime Environment at specified time periods. Tasks can be commands that are executed by Sammi's command processor or functions that are executed by an API peer application. In the latter case, the tasks are specified with commands that are sent to an API peer application at their scheduled times, and the API defines the format and meaning of the command strings. Lists of tasks are generally stored in editable files and loaded when Sammi is started. Task files can be added or deleted during runtime, and can also replace existing task files.

Report Program, s2_report

The report program generates PostScript files for printing Sammi formats. It overloads Sammi's internal low-level drawing procedures with PostScript output procedures. As a result of this overloading, objects that are not widget-based automatically call Postscript graphics procedures instead of X library graphics procedures when s2_report displays them. Since Xt-based widgets do not draw their graphics by calling Sammi's graphics library procedures, Sammi DDOs that encapsulate widgets contain PostScript output methods to approximate their appearance when called by s2_report.

Format Unload Program, fmt_unload

The format unload program is a batch program. It is used in conjunction with the format load program to transfer format files between systems that have different binary file formats and to update formats when Sammi's revision level changes. It generates ASCII-based format files by reading binary format files, then writing their objects to ASCII files as hierarchical structures containing name-value pairs.

Format Load Program, fmt_load

The format load program is a batch program that generates binary format files from ASCII-based format files. This program is used to transfer formats between systems that have different binary data representations and to update formats when Sammi's revision level changes. It scans ASCII-based format files, creates updated binary images of their objects, then writes the binary images to loadable format files.

Utility Programs

The base Sammi ADK contains several useful utility programs that are used to perform functions such as importing and exporting X window dumps, killing Sammi processes, spooling Sammi output files, and various other functions.

SAMMI OBJECTS

As noted earlier, there are two types of objects that are added to Sammi formats: static display objects and dynamic display objects. The following paragraphs describe these objects.

Static Display Objects

Except for composite objects (and classes of objects derived from composites), static display objects are geometric primitives. Their geometries are stored as vectors (bitmapped text is an exception), and they can be transformed with standard homogeneous transform matrices. A variety of line widths and line styles can be used with all of these objects except bitmapped text. Objects whose geometries are closed (rectangles, polygons, closed splines, circles, and ellipses) can be filled with either a solid, opaque color, a transparent color, or a variety of opaque or transparent stipple (fill) patterns. Sammi's static display objects are:

Composites

Composite objects are generalized container objects that maintain and manage a list of child objects. The child objects of composite objects are other composite objects, static display objects, and/or dynamic display objects. They are used to implement the top-level object in a format's object hierarchy, layer objects, groups, and as containers for library parts.

Composite objects can be transformed with rotation, scaling, and translation matrices. When a composite object is transformed, it recursively transforms all of its child objects.

Layers

Layers are composite objects that partition a format's objects into planes, much like stacked sheets of paper. The visibility of individual layers can be controlled with a Sammi command. When a layer is invisible, all of its child objects are invisible.

Groups

Groups are composite objects that are used to "glue" a collection of objects together. They are used to create library parts that consist of multiple objects, and as transient editing objects in the Format Editor.

Lines, Polylines, Rectangles, Polygons, and Splines

These objects are all implemented as lists of points. The point lists for lines, polylines, and polygons are the coordinates of the vertices that connect their edges. The point lists for splines are control points: their vertices are generated dynamically from their control points. Sammi splines are Bezier curves, and their curves do not pass through their control points.

Polylines consist of an arbitrary number of connected points. Lines are polylines that have only two points and one edge, polygons are polylines whose first and last points are equal, and rectangles are constrained polygons. Splines consist of an arbitrary number of control points, and analogous to polylines and polygons, there are both open and closed splines. Open and closed splines can be automatically converted to polylines and polygons, and vice versa, in the Format Editor. With the exception of rectangles and simple lines, points can be added to or removed from these objects, and the points for all of these objects can be individually moved.

Frames

Frames are specialized rectangles that draw their borders in one of four styles: shadow in, shadow out, etched in, or etched out. These are the same border styles as those used by Motif widgets. Frames are generally used to provide a custom border for other objects. The objects are placed on top of a frame, and the two (or more) objects are glued together by "grouping" them.

Arcs

Sammi arcs are three-point arcs: one point defines the center, and the other two points define the starting and ending points on the curve of the arcs. Arcs can be filled or unfilled. Filled arcs can be filled with a "pie" fill style or with a chord fill style. Arcs can be rotated, translated, and/or scaled, and can also be modified by interactively moving their endpoints.

Circles and Ellipses

Both circles and ellipses are defined with a center point, a vertical radius, and a horizontal radius. Circles are constrained so that their two radii are the same length. Both of these objects can be scaled, translated, and rotated. However, since the low-level X library procedure Sammi calls to draw ellipses does not support rotated ellipses, if the rotation angle is not an increment of 90 degrees, they are not drawn rotated.

Text

Sammi supports two types of static text fonts: X bitmap fonts, and Hersey vector fonts. The vector fonts are available in a variety of pre-defined point sizes and styles, and the bitmap fonts are available in all the sizes and styles supported by the X server.

Vector text can be arbitrarily transformed. The bitmap text can not be transformed, but it can be displayed much faster than can the vector text.

Reference Objects and Reference Instances

Reference objects are similar to Sammi library parts - they are composite objects that manage a list of child objects, and are saved as a single, named object. These objects are used as the "bodies" for reference instance objects. Reference instance objects maintain a reference to a named reference object and maintain their own transform information and display attributes. They draw themselves by transforming and displaying their body (reference) object.

Only one copy of a named reference object is maintained in Sammi at any given point in time. Reference objects are read into Sammi's global shared memory space at start-up time, and can be reloaded during runtime. The global copy is shared by all objects that reference it. Modifications made to reference objects are automatically reflected in any objects that reference them as soon as they are loaded or reloaded.

Dynamic Display Objects

Sammi has a rich variety of built-in dynamic display objects, ranging from simple character-based data display and entry fields to multiple plot, trend, and curve display objects.

Most of Sammi's dynamic display objects interact with end users by processing input events. However, some of them are output-only, displaying data graphically, textually, or both. Regardless of whether or not DDOs process input or are output-only, almost all of them support runtime annotations. The exceptions are the container-type objects, the non-display object, and a few of the objects that encapsulate widgets.

Some of Sammi's built-in DDOs encapsulate Motif widgets and process events with X Toolkit callback procedures, but most of them process events and display their graphics through Sammi's event handlers and graphics interface library.

Popup DDOs such as menus and select lists can be attached to other Sammi DDOs and are automatically popped up with a mouse button click. These popups can send commands to Sammi, to API applications, or to other DDOs, and can also write values into the DDOs to which they are attached, if these DDOs are character-oriented. Additionally, popup help windows can also be attached to Sammi DDOs and automatically popped up to display the contents of a named help file.

The following paragraphs describe Sammi's built-in dynamic display objects.

Character-Oriented Input/Output Fields

This group of DDOs displays their runtime data as ASCII text strings, and end users type values into them. They can be created as output-only objects, and can also be dynamically changed from input/output objects to output-only, or vice versa. When a new value is entered in an integer, real, or time DDO, the value is converted from a string to a binary value before it is sent to the DDO's data source. The runtime data for the ASCII, text browser, and text field DDOs are simply character values, and thus do not require conversion. The runtime values for these DDOs can consist of datasets, which are multiple records (rows) of values. The input/output fields are:

- *ASCII* - this DDO is a simple, variable-length text field.
- *integer* - the integer DDO accepts short or long integer values as its runtime data. These values can be displayed in a variety of formats: decimal, hexadecimal, octal, with or without leading zeros, and shifted and/or masked.
- *real* - the real DDO processes floating point numbers. The runtime data for these objects are either single- or double-precision binary floating point values. The real DDO displays its values in either fixed-point or scientific notation.
- *text browser* - the text browser DDO encapsulates a Motif Text widget and can be used as a simple file editor or file browser. Its runtime data can consist of actual ASCII character data or a file name. Its runtime data can also be statically defined when the browser is created.
- *text field* - the text field DDO encapsulates a Motif TextField widget. It is generally used for simple one-line text entry and display.
- *time* - the time DDO displays binary time values in a user-defined display format, and converts user-entered time values to binary values. It supports all 26 major time zones and daylight savings time conversions. It also supports customization of time-related names (days, months, time zone names and abbreviations, and AM and PM abbreviations).

Character-Oriented Output-Only Fields

Sammi has several built-in output-only DDOs that display text strings using display formats that are defined when the DDOs are created in the Format Editor. They are:

- *alarm field* - this DDO is used by Sammi's alarm process to format and display alarm and event data.
- *annotated text* - the annotated text DDO processes character data containing optional, embedded formatting (annotation) characters. The embedded formatting characters allow colors and fonts for individual characters or words in the displayed text string to be dynamically modified during runtime.
- *equation* - the equation DDO evaluates expressions and displays the result as a string. It reads values from other DDOs and uses the values in the expressions. The expressions recognized by the equation DDO consist of a subset of the C programming language, including function calls. These expressions are internally compiled before they are evaluated and displayed.
- *formatted numeric* - the formatted numeric DDO accepts short, long, float, and double precision binary values, processes the values using the equation DDO's expression evaluator, and displays the results. The expression used to process and display the numeric values can be dynamically changed during runtime. The formatted numeric DDO is primarily used to perform and display unit conversions (i.e., centimeters to inches).
- *tabular* - this DDO formats and displays values in a tabular (row, column) format. The runtime data for the tabular DDO consists of structured records. The records can contain combinations of characters, short and long binary integers, single- and double-precision binary reals, and variable-length ASCII strings. The record layouts (field names and types) are described in text files, and the display format and column titles are described with a formatting string. The record descriptions are similar to ASCII record descriptions supported by most database management systems, and the formatting strings are similar to C formatting strings.

Status Objects

Sammi has two built-in DDOs that accept binary numeric values and use the values to look up and display either an ASCII string or a bitmap symbol. The text strings and symbols are defined in lookup tables that map values to table entries. The table entries are either ASCII strings or names of bitmap symbols. These DDOs support various indexing schemes to determine which table entry to display. The table entries typically indicate the status of some resource or entity, such as a fuel level or whether or not a valve is open or closed.

- *text table* - this DDO uses binary numeric values to look up and display an ASCII string from a table. Examples of some table entries are "on", "off", "open", and "closed".
- *symbol table* - this DDO uses binary numeric values to look up and display a cached bitmap symbol. Examples of some symbols are bitmap images depicting switches in various positions.

Command Objects

- *transparent button* - transparent button DDOs are rectangular regions that briefly "flash" and send a command when a mouse button press occurs inside them. They are used as "hot spots" and can be placed anywhere in a format. They are generally placed over text, bitmaps, or vector images.
- *pushbutton and toggle button* - these DDOs encapsulate Motif buttons. They can contain text labels or bitmap images.
- *menu* - Sammi supports both pull down and popup menus. Each type can include an arbitrary number of cascading (pull right) submenus. The menu items can be either pushbuttons or toggle buttons, and can use text labels or bitmap images. Menus can include separators and titles, and they support use of keyboard pneumonics. The menu items and corresponding commands for menus can be defined statically in the Format Editor, in an ASCII file, or dynamically at runtime by an API application.
- *option menu* - the option menu DDO encapsulates a Motif OptionMenu widget. This DDO is similar to a pushbutton with a text label, but when it is selected, it pops up a menu list. When an item is selected from the menu, its text string is displayed as the pushbutton's label.
- *select list* - the select list DDO encapsulates a Motif list widget. It supports both single and multiple item selection, and can be created as either a popup or non-popup list. The list items and corresponding commands for select lists can be defined statically in the Format Editor, in an ASCII file, or dynamically at runtime by an API application.
- *object icon* - the object icon DDO is similar to a button. However, instead of one or two states, the object icon supports multiple states. Each time a button press occurs in an object icon, it sends a command, changes state, and displays a bitmap image that depicts its new state.

Chart and Analog Readout Objects

Sammi has several built-in DDOs that model analog devices or graphically display dynamically changing data as charts. These objects are:

- *bar graph and bargraph* - the bar DDO displays runtime values as a horizontal or vertical filled rectangle. As the bar's runtime value changes, the height or width of the filled rectangle in the bar increases or decreases. The bargraph displays multiple bars arrayed along either a vertical or horizontal axis. The bargraph DDO is used to display multiple, changing values. Up to 256 bars can be displayed at the same time by a bargraph.
- *piechart* - the piechart DDO is a standard circular piechart containing filled arcs. The arcs are filled with different types of fill patterns and/or colors, and their sizes reflect different values as percentages of an overall value. The sizes of the piechart DDO's filled arcs change in response to changes in its runtime data values.
- *legend* - the legend DDO is used with Sammi's chart and graph DDOs to indicate the meaning of colors, line and fill styles, and markers.
- *meter and gauge* - These two DDOs model actual meters and gauges: they contain a needle whose position

changes as their runtime data values change. Both of these DDOs support a variety of optional display characteristics for indicating limits, warning, and critical values.

- *moving-scale and fixed-scale linear meter* - like meters and gauges, these two meters model actual analog devices. Each contains values, but instead of a needle, these meters contain a triangular value indicator. In the moving-scale linear meter, the indicator remains in the center of the meter, and the displayed values move. In the fixed-scale linear meter, the values remain fixed but the indicator moves. Each of these meters can be displayed horizontally or vertically, and each supports a variety of options for indicating limits, warning, and critical values.

Controller Objects

Three of Sammi's DDOs are used to control how other DDOs display their values, to set values for other DDOs, or to move, scale, and rotate both static display objects and other DDOs. They are:

- *scrollbar* - the scrollbar DDO encapsulates a Motif scrollbar widget. It is generally used to scroll datasets of input/output objects. Its runtime data consists of a current scroll value, scroll minimum and maximum values, and scroll increment. When end users move the scrollbar's slider, the scrollbar sends its new current value to API applications, which then update the runtime data for the scrolled target DDOs.
- *slider* - this DDO encapsulates a Motif scale widget. It is generally used to set values for meters, gauges, or any other DDOs that display numeric data. The target values are updated in the same way the target DDOs for scrollbars are scrolled: the slider sends its new value to the API application, which then updates the runtime values for the target DDOs.
- *dynamic object* - the dynamic object DDO is used to transform and/or modify the display attributes of other objects. It does this by looking up transform values (scale, rotation, translation) and display attributes (color, line style, etc) in a table, using its runtime data value as the lookup index. This object uses double-buffering to avoid screen flicker when it erases and redraws its controlled objects.

Widget Container Objects

Sammi provides three widget-based DDOs that are used as containers for other DDOs that encapsulate widgets. They are:

- *button group* - this DDO encapsulates a Motif button group widget and is used as a container for Sammi's pushbutton and toggle button DDOs.
- *paned window* - this DDO encapsulates a Motif paned window widget. It can be used as a container for any of Sammi's widget-based DDOs.
- *tool bar* - the tool bar DDO encapsulates a Motif frame widget containing a row/column widget. It can be placed in a Sammi format as a vertical or horizontal region and attached to the top, bottom, left, or right edge of the format. Its position and size in the format remains fixed when the format's contents are panned and/or zoomed.

Graph Objects

Sammi has three DDOs that are used to plot coordinate values. The runtime data values for each of these DDOs consist of curve data, which are groups of coordinate values (an *x* and *y*, or *time* and *y* value) and optionally, a coordinate point quality value. They can be used to plot a fixed range of changing values, or plot changing values whose range also dynamically changes.

The graph objects support various options that control how (and if) labels, axes, grids, markers, and titles are displayed. They also support display of multiple curves in different colors, line styles, and line widths. The displayed graphs can be panned, zoomed, and ranged during runtime. Additionally, the plots can be cleared and reset, and their display attributes can be modified at runtime with a "change attribute" command. The graph objects are:

- *plot* - the plot DDO displays curves whose coordinates are (*x*, *y*) pairs and an optional quality value for each coordinate.

- *trend* - the trend DDO displays curves whose coordinates are (time, y) pairs and an optional quality value for each coordinate.
- *curvesets* - the curveset DDO is used to plot multiple-valued data as sets of curves in a single graph. This DDO supports display of up to 256 curves.

Miscellaneous Objects

Several DDOs are used for miscellaneous purposes. They are:

- *graphical data field* - this object displays an image stored in a file. Its runtime data consists of the name of the file containing the image data. The image formats supported are X bitmaps and pixmaps, GIF images and XWD images.
- *no-display object* - this object does not display any graphics or text. It allows API applications to associate miscellaneous information with a format via its read and write key fields, and obtain the information at runtime.
- *region* - the region DDO is a rectangular area that API applications can use as a drawing area in a format during runtime.

SAMMI COMMANDS

Commands are primary application and Sammi control constructs: they result in application and/or Sammi actions. API applications send commands to the Sammi runtime and/or other API applications, and Sammi sends commands to API applications. Additionally, users can issue commands by typing them in the command line of the Runtime Environment's command window or the Format Editor's command window. As previously discussed, commands can be placed in user command files and automatically executed when a user logs on to Sammi.

The following paragraphs describe the most frequently-used built-in Sammi commands. The Format Editor also has a large number of commands that are specific to it. The commands unique to the Format Editor are not described here.

Format Display Commands

This category of commands control which formats are displayed. These commands are:

- *add-window* - adds a format to a screen
- *delete-window* - removes a format from a screen
- *pop-window* - makes a format the topmost visible format on a screen.
- *purge* - removes all deleted formats from Sammi's internal format cache.
- *push-window* - lowers a format so that it is beneath other formats on a screen.
- *recall-window* - redisplay a deleted format at its previous position and size.
- *replace-window* - replaces a displayed format with another format, positioning and sizing the new format at the same location and size as the replaced format.

Format Paging Commands

Each Sammi format contains six fields that are used to chain a sequence of formats together into pages. The fields are: first page, last page, page backward, page forward, page up, and page down. The field entries are format names.

The format paging commands enable API applications or end-users to page through the chained formats. Each of these commands removes the currently displayed format from the screen and adds the format whose name is in the corresponding page field of the removed format. The paging commands are:

- *first-page*
- *last-page*
- *page-backward*
- *page-forward*
- *page-up*
- *page-down*

Drawing Control Commands

Several Sammi commands control which objects in a format are visible and how they are redrawn. These commands are:

- *declutter* - the declutter command enables or disables the display of objects based on a declutter attribute defined when they are created. This attribute is used when a format is re-scaled (zoomed), and specifies upper and lower scale redisplay limits for an object. When the object is scaled so that its size is outside the range of its declutter limits, it does not display itself.

- *expose* - forces redisplay of an entire format or redisplay of specified objects.
- *pan* - scrolls a format's objects by translating their position coordinates left, right, up, or down.
- *refresh* - forces polled-update DDOs to request data from their servers and redisplay themselves with their new runtime values.
- *visible* - toggles the visibility attribute of a format's layers or objects. This command makes layers or objects visible or invisible.
- *zoom* - scales a format's viewport so viewable objects appear larger (zoom in) or smaller (zoom out).

DDO Commands

The following group of commands are used to modify various DDO attributes:

- *change-dfd-attribute* - this command is used to change the enter-ability attribute of input DDOs and to change various display attributes of Sammi's plot, trend, and curveset DDOs.
- *graph* - controls how Sammi's plot, trend, and curveset DDOs respond to a zoom command.
- *popup* - this command pops up a popup DDO.
- *set-sensitive* - this command toggles the sensitivity of Sammi DDOs that process keyboard input events and/or mouse button events. When an object is made insensitive with this command, it displays its text with a "ghosted" appearance to indicate it is not sensitive to input.

Session Commands

As discussed in an earlier paragraph, sessions are groups of runtime keys that are associated with a format, and runtime keys are pairs of named variables and values. Runtime keys are used in command strings and in various names, and Sammi's command dispatcher substitutes values for the runtime key variables before it dispatches the commands. The following commands are used to bind or rebind values to runtime key variables, to establish sessions, and to bind a session to a format:

- *format-session* - this command binds or unbinds a session to or from a format, or reports the name of the session that is bound to a format.
- *session* - this command creates a named transient or permanent session. Transient sessions exist only as long as they are bound to one or more formats, and permanent sessions exist even when they are unbound.
- *set-runtime-keys* - this command binds or unbinds a value to or from a runtime key (named variable), and optionally, assigns the runtime key to a named session. The runtime key can be a global variable or a session variable.

Network-Related Commands

Sammi maintains an internal table that maps logical server names to RPC numbers and network node ids, and stores other information about logical servers. This information is generally defined in a Sammi configuration file and used to construct the internal table when Sammi starts up. The following three commands are used to modify Sammi's internal network configuration table, to execute commands from a local Sammi system on a remote Sammi system, and to control synchronization between Sammi and API applications.

- *logical-server* - this is a general command for adding and removing logical servers, connecting to, reconnecting to, and disconnecting from them, changing their parameters, and establishing their timeout intervals.
- *remote-command* - this command is used to execute a Sammi command entered in a local Sammi system on a remote Sammi system.
- *sync* - this command forces Sammi to resynchronize with API applications or establishes the rate at which

Sammi synchronizes with them.

Process and file-related commands

The following group of commands are used to read various Sammi files, set Sammi and Format Editor default values, or to execute Sammi processes and utility programs. These commands are:

- *hard-copy* - this command prints an X Window Dump image of a screen or a Sammi format.
- *load-defaults* - the load-defaults command is used to load or reload default values from a named defaults (preferences) file. Defaults files contain name-value pairs that set values for many Runtime Environment and Format Editor attributes.
- *read-command-file* - reads a named command file and executes the commands in sequence.
- *read-uaf-file* - the read-uaf-file rereads a user authorization file.
- *redefine-keys* - this command reads a keyboard bindings file and re-maps the key definitions.
- *report* - generates and prints a Postscript plot of a format.
- *schedule* - this command is used to manipulate the task schedule lists used by Sammi's task scheduler. These lists contain tasks that are to be performed at scheduled times.
- *set-default* - used to set a single Runtime Environment or Format Editor preference value.

System Commands

- *alias* - assigns an alias (usually a short name) to a command or group of commands.
- *logoff* - ends a user's Sammi or Format Editor session and deletes all formats except permanent formats (i.e., the Sammi command window).
- *logon* - starts a user's Sammi or Format Editor session. When a user logs on, the user's command file is read and executed, and the user's keyboard bindings file is read and the key bindings are modified. The user command file may contain a load-defaults command to load and set the user's preferences.
- *password* - this command is used to define a password, providing the user has authorization to use this command.
- *quit* - ends a user's Sammi session, deletes all formats, and shuts down the Runtime Environment or Format Editor.
- *system* - this command executes an operating system command.
- *unalias* - removes an alias.

Miscellaneous Commands

- *trace* - this command turns diagnostic trace output on or off. The diagnostic output is saved in a file.

MISCELLANEOUS SAMMI FEATURES

Application Redundancy (Failover)

Sammi supports designation of API applications as redundant for other API applications. A redundant application is a backup application: it automatically assumes the functions of another application should it fail. Sammi's network management layer automatically reroutes connections between Sammi and an application's designated backup application when Sammi loses the network connection to the failed application.

Applications are made redundant by simply assigning the same logical server name to multiple applications. The redundant applications can be on the same or different network nodes. Sammi's runtime processes maintain an internal table of logical servers, and when a server fails, Sammi simply looks for the next entry in the table with the same logical server name as the failed server, then connects to it.

Security

Sammi includes logon security and facilities for limiting access to specific Sammi functions, formats, and DDOs. As examples, some users may be authorized to log on to Sammi's Runtime Environment, but not to the Format Editor; some users may be restricted from changing configuration files; some users may be restricted from adding certain formats or entering values into certain DDOs; and some users may be restricted from even viewing certain DDOs.

Access privileges are established by system administrators with user authorization files and security class files. The user authorization file contains entries for each Sammi user. A user authorization entry consists of user name, password, and security class. It may also include the name of a keyboard binding file, a Sammi command file, and a time range within which a user can log on to Sammi and/or the Format Editor. If keyboard binding and command file names are present, when the user logs on to Sammi, the commands in the command file are executed and the keyboard is remapped using values from the keyboard bindings file. The user's security class entry is matched with an entry in the security class file to determine the user's access privileges. Sammi supports up to 255 security classes, and each class can contain up to 28 different access levels.

Each format and DDO contains a security class field. Security class values are placed into these fields when formats and DDOs are created, or default values can be used. When a format is added to the screen, Sammi looks at these values and compares them to the security class of the user who is currently logged on to determine if the user is authorized to add the format. If the user is authorized to add the format, Sammi then looks at the DDO security class entries to determine the user's individual DDO read/write privileges.

Multiple Data Paths

Sammi format, library part, and data files are stored in system directories. To enable partitioning of all these files into logically related groups, Sammi supports multiple data path specifications. A Sammi data path is simply a string containing a directory specification. These paths can be concatenated to form an ordered search path through a file system and placed in Sammi's configuration file. Sammi then uses the search path to locate its files.

Application development groups can use multiple data paths to partition the components of their overall application by revision, by customer, or by any other criteria that facilitates their work.

Table Editors

Almost all of Sammi's built-in DDOs access values from user-defined runtime annotation tables that specify color and blink attributes. Additionally, Sammi's symbol table, text table, and dynamic object DDOs access object-specific values from user-defined tables. These tables are stored in ASCII files, and can all be created with any text editor. However, the table entries must conform to a specified format, and entries must follow a specified order. As a result, it is easy for users to make errors when creating these files. Sammi's table editors address this problem.

The Sammi table editors interact with users to obtain table information and table entries, validate the entered data, then generate or update the ASCII files for the target tables.

Read/Write Key Editor

Sammi uses the logical server name stored in a DDO to determine which API application is its data source/sync, and API applications use its read and write key strings to determine various information for or about the DDO. These values must be correct for an application to manage a DDO. End users enter these values when DDOs are created in the Format Editor. Although erroneous entries can be readily detected, the Format Editor provides an editor that is used to examine and edit the server names and read/write keys for all DDOs on a format.

API / USER INTERFACE INTERACTION

API applications interact with their user interface components via data values, commands, and API event messages. API applications send data and commands to their user interface components, and the Sammi runtime processes send data, requests for data, commands, and events to API applications. For this interaction to occur, the Sammi runtime processes need to know which API process is driving a particular user interface component, and the API process needs to know which formats and user interface components are active.

Process and User Interface Component Identifiers

Every DDO in a Sammi format contains a field that identifies the API process responsible for the DDO. This identifier is a string called a logical server name. Logical server names are assigned to DDOs by format designers when they create them with the Format Editor.

At the network level, RPC numbers identify Sammi runtime and API processes. System administrators assign logical server names to RPC numbers in a Sammi configuration file that is processed when Sammi is started, and make the names available to API programmers and format designers. The API programmers and format designers can thus view the network simply as a collection of named processes.

When a format is added to a screen at runtime, Sammi constructs lists of DDOs organized by server IDs, and assigns each DDO a unique identifier (an integer number). Sammi then sends these identifiers to the API applications. Format names and DDO IDs are used by both Sammi runtime and API applications to identify specific user interface components, and server ids are used to route data between the components and their controlling application processes.

Associative Key Identifiers

In addition to logical server name fields, DDOs also have read key and write key fields. These fields are also strings, and they are provided so that arbitrary identifiers (keys) and/or other information significant to an application can be associated with specific DDOs. Read and write keys are not used by Sammi's runtime processes. They are simply sent to their DDO's controlling process by the API along with other information about the DDO.

For example, in database applications, read and write keys generally correspond to actual database key values, identifying records and fields containing values for DDOs. However, applications can use them for any purpose. They can be used to symbolically specify a sequence of actions that an application should perform when an event occurs in a DDO, or to identify another DDO or group of DDOs that require updating when a DDO's data values change.

Sammi/API Data Transfer

At the simplest level, API applications and Sammi runtimes interact with each other and with end users by reading and writing data values (runtime data). When a format is added to a screen (by an end user, a command sent by a DDO, or a command sent by an API application), Sammi runtime requests data values for all the DDOs in the format. The API application receives the request and uses the DDO identifiers, along with their read/write keys, to determine what data to return, then sends the data back to Sammi. Sammi then initializes the DDOs with their data values and displays them. As end users interactively edit and modify values through various input techniques (such as keyboard entry or mouse 'clicks' and 'drags'), Sammi sends the modified values back to the API application, which then decides what action to take based on the end user's modifications. This reading and writing of runtime data continues until the format is removed from the screen, at which time Sammi notifies the application, letting it know which format was removed and which DDOs are no longer active.

The data transferred between an application and the Sammi runtime processes consist of either simple values, such as integers, reals, or text strings, or of structures of values, such as plot points. Single or multiple data points (called datasets) can be transferred in a single network transaction.

API applications can associate quality values with data points. Quality values are attributes of the associated data points, and are called runtime data annotations. They modify how DDOs graphically display their runtime data. Quality values generally indicate limits or critical values, and usually result in their associated data being displayed in special colors, with special appended characters, blinking, or any combination of these three attributes. Quality

values are transferred along with their associated data points.

For some applications (such as process monitoring or data entry applications), reading, writing, and updating data values as described above is almost all the API/Sammi interaction that is required. However, API applications (including the two types just mentioned), also need to respond to end users in ways other than simply displaying data values. The primary software control construct in Sammi applications is a command.

Sammi Commands

Both the Format Editor and the Sammi runtime programs process Sammi commands. These are structures that contain a command sender, a command receiver or target, and a command string that identifies an action and specifies parameters for the action. An example of a Sammi command is the string "add-window my_format". This command adds the format named my_format to the screen, if it is found. When Sammi's command dispatcher receives a command, it looks for the receiver of the command, and if it finds the receiver, it sends the command string to it. The recipient of the example add-window command is Sammi's local command processor, which parses the command string, looks up the command procedure that is responsible for the command, then calls the procedure, sending it the command parameters, if any.

Commands are sent by or to API applications, Sammi's local command processor, procedures that manage windows, or DDOs. End users can issue commands directly to Sammi's local command processor by typing them in Sammi's (or the Format Editor's) command window. Many commands that implement a variety of common application actions are an integral part of Sammi. These built-in commands are listed and described in a later section.

Issuing commands is the primary input event action of some Sammi DDOs. These DDOs are called command DDOs. The most frequently used command DDOs are pushbuttons, select lists, and menus, which send commands in response to mouse button events. A single command can be attached to a pushbutton and two commands ("on" and "off") can be attached to a toggle button. For menus and select lists, commands can be attached to each menu entry or select list item. Commands can be defined and attached to command DDOs in several ways: by defining them in the Format Editor when the DDOs are created or edited; by defining them in an API application and sending them to the DDOs as runtime data; or by defining them in a file which the DDOs read during runtime.

Since command DDOs automatically send commands, and Sammi's command processor dispatches or executes commands without requiring any intervention by API applications, many common application actions can be implemented with commands instead of with code. Liberal use of Sammi's command processing not only reduces code volume, it also reduces the volume of network traffic in a Sammi application.

To increase the generality and usefulness of commands, Sammi supports the use of symbolic variables and aliases in commands. The command processor replaces the symbolic variables and aliases with their values before it dispatches or processes a command. An alias is simply a short name or abbreviation that replaces an entire command string, including command arguments. The symbolic variables are called runtime keys.

Runtime Keys and Sessions

A runtime key is a Sammi runtime variable containing a string value. The runtime key's string value is substituted at runtime in DDO read/write key strings, logical server name strings, and command strings that contain the specified runtime key variable. Values are associated with runtime keys with a "set-rtkey" command. An example is "set-rtkey next_format my_format", which assigns the string "my_format" to the runtime key "next_format". This runtime key can then be used in a command to add a format to the screen as follows: "add-window @next_format". The '@' character distinguishes runtime key variables from standard strings.

Runtime keys allow API applications to change the way they interact with Sammi and DDOs during runtime. As examples, a command attached to a pushbutton, such as the add-window command in the preceding paragraph can be modified by changing it with a runtime key; and a DDO's data source can be changed by changing the name of its logical server with a runtime key.

Runtime key variables and runtime key values can be grouped into sets using sessions. A session is a named group of runtime key/runtime value pairs that are bound to and used for a specified format. Sessions are created using an optional session name with the set-runtime-keys command. To illustrate, the following command groups two runtime keys, key1 and key2, in a session named my_session: set-runtime-keys -s my_session key1 "hello" key2 "world".

Since different values can be assigned to runtime keys, multiple sessions can be created from a single set of runtime keys. Thus, a single format containing a fixed set of runtime keys can be dynamically modified by simply changing its session. This is done with a `format-session` command. For example, to bind the session, `my_session` (from the preceding paragraph), to a format named `my-format`, the command is: `format-session my_format my_session`.

API EVENTS

API-based applications and data servers are event-driven - they wait for an event message from the Sammi runtime processes, respond to the message, then wait for another message. Sammi's API event messages and the data structures associated with them are defined in include files, and specify the types of messages API applications and servers receive from and send to the Sammi runtime processes.

The API event messages are identified by an event type code, which is an integer number. The API include files contain macros (`#defines`) for the event types. API applications/servers use these event types to register message callback procedures with the API event input procedure, or use them as message identifiers in an event input loop if they are managing their own input.

The API defines fourteen different event messages. The API event types and their meanings are discussed in the following paragraphs.

Connection and Input Timeout Events

Two event codes are used with Sammi's interprocess communications facilities. These two event codes are:

- **S2_ACTIVATE** - This is the first event a Sammi runtime sends an API application. It is sent the first time a reference to an application is encountered and Sammi establishes an IPC connection with the application. This typically occurs when a format containing a reference to an API application is added to the screen. It can also occur when a command to change a logical server is issued, or when a logical server fails and the application is the failover (backup) server for the logical server that failed.
- **S2_TIMEOUT** - This event occurs when the API event input procedure times out; i.e., no input event message is received within a specified time interval.

Format Events

The following event types are used to notify an API application/server when a format containing references to it is added to or removed from the screen:

- **S2_ADD_WIN** - Sammi sends the **S2_ADD_WIN** event when a format is added to the screen and the application/server is the logical server for one or more DDOs in the format. The data associated with the event contains a list of the DDOs that reference the application/server. The list contains information about the DDOs (for example, their read/write keys).
- **S2_DEL_WIN** - This event is sent when a format containing DDOs that reference an application/server is deleted from the screen.
- **S2_VADD_WIN** - Sammi sends this event when a format is added to the screen and the application/server is defined as the format's logical server.
- **S2_VDEL_WIN** - *the S2_VDEL_WIN* event is sent to the application/server that is defined as the logical server for a format when the format is deleted from the screen.

Runtime Data Events

There are three event types used in conjunction with runtime data request and update messages. They are:

- **S2_READ_DFD** - This event is a request for runtime data for DDOs that are periodically updated. It is sent to the application/server specified as the logical server for a polled-refresh DDO when its refresh (update) interval expires.
- **S2_REPLY** - This event is used to notify an application/server that an enqueued runtime data update request has been completed.
- **S2_WRITE_DATA** - The **S2_WRITE_DATA** event is sent to an application/server to notify it that the runtime data for a DDO it is driving has been modified. The data associated with this event message contains the

new value(s) for the DDO.

Dynamic Display Object Events

Several event type codes are used to notify an API application/server about mouse button events that occur in its DDOs or about miscellaneous changes to its DDOs. They are:

- *S2_BTN_EVENT* - This event is used to send a command from an option popup DDO to an application/server. Option popup DDOs are select lists or menus that are attached to another DDO and popped up when a user selects the DDO with mouse button 3. When an item from the popup list or menu is selected, Sammi uses this event to send the command associated with the selected item to the logical server designated to receive the command.
- *S2_CHG_DFD* - This event is used to notify an API application about miscellaneous changes in one of its DDOs. The event message is sent when the visibility of a DDO is changed, its read and/or write keys are changed because a runtime key is changed, or the DDO is scrolled and its viewable contents are changed.
- *S2_SELECT_DFD* - This event is used to notify an API application when a mouse button release event occurs in one of its DDOs. This event is optional, and is not sent unless specifically enabled on a per DDO basis.

Miscellaneous Events

- *S2_ACK_EVENT* - This event is sent by a Sammi runtime's alarm process when a user acknowledges an alarm that was previously issued by an API application.
- *S2_SEND_CMD* - The *S2_SEND_CMD* event is used to let an API application know it has received a command, either from a command DDO or from another API application.

CUSTOMIZING SAMMI

Although Sammi has a wide variety of built-in functionality and a rich set of DDOs, many application domains require functionality and/or objects that are not available in Sammi. The Sammi Developer's Kit is a toolkit that enables programmers to extend, enhance, and modify Sammi and the Format Editor to meet their system's requirements. However, both the Sammi and Format Editor user interfaces and library components can be customized even without the SDK.

Customizing Sammi without the SDK

Since Sammi's user interface components (command windows, palettes, toolbars, etc) and built-in library components are constructed with the Format Editor, they can also be modified with the Format Editor. This level of customization does not require any programming. Format designers can:

- remove unwanted or unneeded functionality from Sammi or the Format Editor by simply removing command objects (buttons, lists, menus, etc) from their formats;
- add command objects to Sammi/Format Editor formats that execute built-in Sammi commands;
- modify the appearance of the Sammi/Format Editor formats by rearranging components within a format, changing colors and fonts, changing bitmaps and pixmaps, etc;
- modify, create and add library components to the Format Editor by editing Sammi's built-in library parts and by creating and adding libraries of composite parts and reference parts;
- create and attach help files to any user interface components to provide descriptions of valid input data, restrictions, etc;
- create different defaults (preference) files for different applications and/or users;
- create different runtime configurations for different applications and/or users by copying and changing Sammi/Format Editor configuration files; and
- partition formats by applications or categories using different data paths.

One use of the above customization capabilities is to simplify the format layout process and to reduce format design errors. This can be done with very little effort. For example, if an application design group knows that only certain logical server names are valid, the server names, descriptions, and restrictions can be placed in a help file and/or in a popup select list or menu. The popup list or menu and the name of the help file can then be attached to the logical server name entry field in the DDO data access definition format, providing format designers immediate access to logical server information and direct button-click entry of the logical server name.

Customizing Sammi with the SDK

Programmers use the Sammi Developer's Kit to customize Sammi and the Format Editor at the code level. SDK programmers can:

- add new commands to the Runtime Environment and Format Editor by implementing command procedures;
- add Window Application Procedures (WAPs) to the Runtime Environment and Format Editor to manage specific formats and manage the interaction between end users and the objects in the formats;
- add new DDOs to the Runtime Environment and Format Editor.

Commands provide the system-level software control for the Runtime Environment and the Format Editor. Command procedures are software modules that implement a command's actions. SDK programmers can create commands that interact with the Runtime Environment, the Format Editor, and with API peer applications and data servers. An example of a command an SDK programmer might implement is a Format Editor command to extract and list all logical server names and runtime keys used in a format that is being edited.

Window Application Procedures are software modules that manage formats. These procedures implement the top-

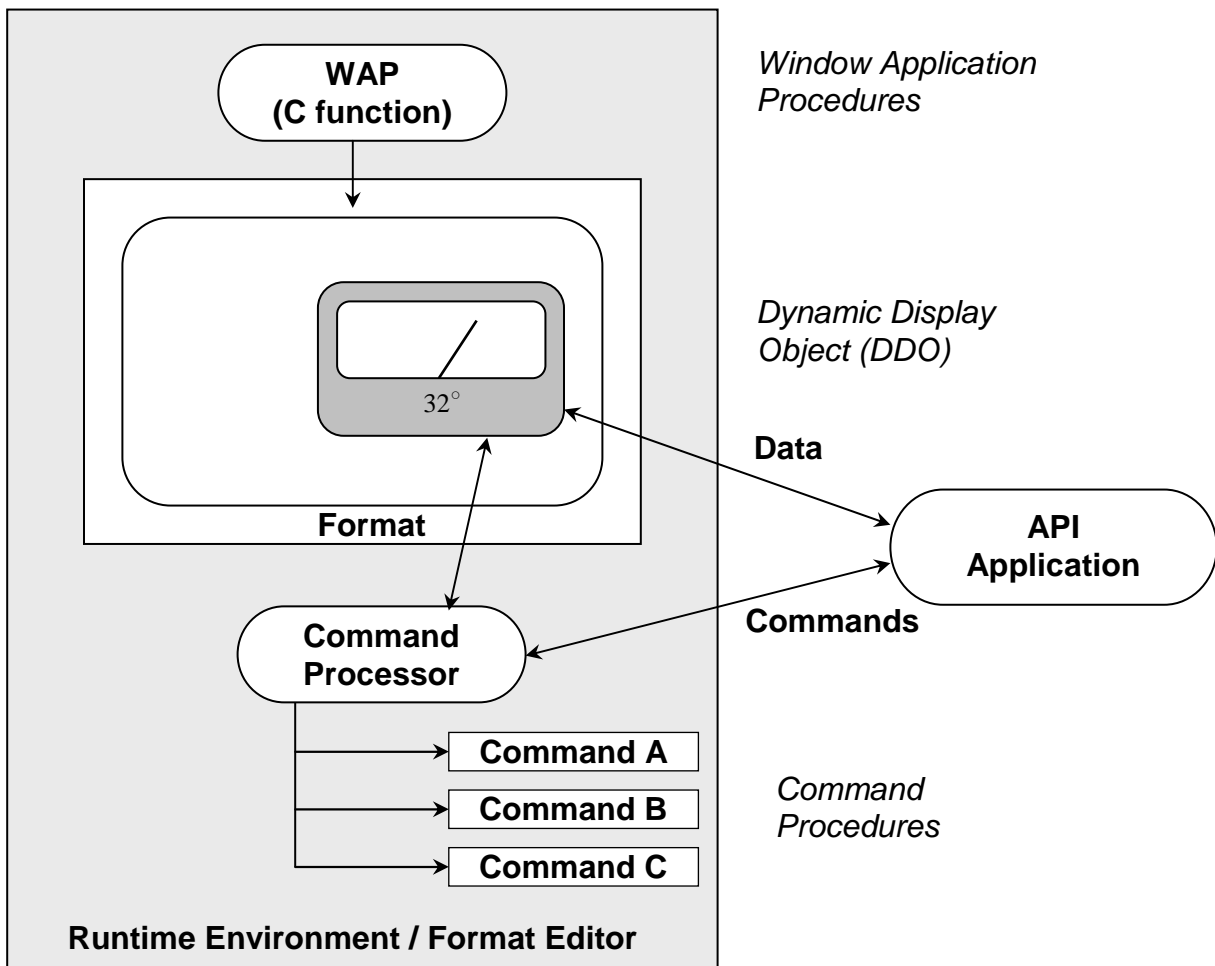
level management and control of all the objects in a format, and manage user interaction with a format's objects. They are also the data sources for DDOs whose data access type is "local". WAPs are generally used to implement custom dialog windows.

WAPs are similar to API peer applications: they receive similar events, send and receive Sammi commands, and read and write runtime data values from and to DDOs. The primary differences between WAPs and API applications are that WAPs are internal to Sammi and the Format Editor, and they manage single formats instead of an entire Sammi application. Since WAPs are internal to Sammi, they can access internal data structures, call Sammi library procedures, or call any other procedures, including X library, X Toolkit, and widget procedures.

An example of how Sammi's built-in WAPs are used is provided by the Format Editor's DDO definition formats. Each of these formats is managed by a WAP that encapsulates information about a specific class of Sammi objects, and each is responsible for interacting with format designers to obtain values for their objects. Another example is provided by Sammi's logon format, which is responsible for obtaining a user's identity and password, then determining if the user is authorized to use Sammi.

DDOs are the graphical user interface components of a format. They display data, process commands, and handle interaction with users. SDK programmers can modify existing DDOs or create new ones. These new DDOs can be implemented from scratch, or encapsulate existing user interface components such as X Toolkit-based widgets. The following figure illustrates how DDOs, WAPs, and commands interact with each other and with API applications:

Figure 12.: DDOs, WAPs, and Command Procedures



SDK Components

The SDK provides access to Sammi's link libraries and object-oriented development framework and associated tools. The major components of the SDK are:

- a set of foundation classes from which new classes of objects are derived;
- a class definition tool that automates the definition of new classes of objects, and partially automates their implementation;
- a configuration tool that automatically binds SDK-implemented commands, WAPs, and DDOs to Sammi's runtime programs and Format Editor and rebuilds the executable programs.
- Sammi link libraries and object files.
- Extensive documentation describing Sammi's software architecture; how software is developed with the SDK; how SDK-developed software interacts with Sammi; what library procedures are available to SDK developers; and how the library procedures are used.
- An extensive collection of contributed SDK-developed software that can be used "as is", or used as examples.

Foundation Classes

Sammi's object-oriented software development framework provides the functions (methods) and structures (classes and objects) to implement DDOs. It consists of:

- procedures, called class interface procedures, that constitute the interface between internal Sammi code and DDOs; and
- extensible classes that implement basic, inheritable object functionality. All DDOs are derived from these base classes.

Sammi's class interface procedures are analogous to C++ envelope classes and to the X Toolkit's "Intrinsics" procedures. They implement the messaging protocol and polymorphic functionality for Sammi objects; implement a type of inheritance known as "chained" inheritance; and implement the constructors and destructors for Sammi objects.

Sammi classes are organized in a single-inheritance hierarchy. The root class is called "object" class, and all other classes are derived from it. The actual instance classes, such as "meter" class, are the leaves in the class hierarchy. There are several intermediate classes between the root class and the instance classes. These intermediate classes extend and/or specialize their base classes, and partition their derived classes into various categories. Since new classes of objects are derived from Sammi's base and intermediate classes, these classes constitute a set of framework classes that specify and define:

- how DDOs are organized;
- what data elements DDO object structures must contain;
- which methods a class of DDOs must implement and which methods can be inherited; and
- what actions a DDO's methods must implement.

Class Definition Tool

Manually creating and initializing the various structures to implement a class of objects is tedious and error-prone, and manually updating the structures if their base classes are modified is problematic. The Sammi Class Definition Tool (CDT) solves these problems. The CDT consists of a macro-like Class Definition language (CDL) that is used to define classes of objects, a CDL compiler that automatically generates class and object structures and initialization code from CDL statements, and procedures for binding new classes of objects to the Runtime Environment and Format Editor. This tool:

- simplifies the task of defining classes;
- generates data structures that conform to the Sammi object-oriented development framework;
- updates dependent classes when their base classes change; and
- implements direct method inheritance.

SDK programmers define the data structures and procedures for a class of objects with CDL statements and save them in a text file. The files are processed by the CDL compiler to generate C data structures, include files, and skeleton procedures for the new classes of objects. After fleshing out the procedures, the new classes are bound into the Format Editor and runtime programs with the Kinesix Reconfiguration Utility (KRU), an object-oriented configuration tool.

Kinesix Reconfiguration Utility

The Kinesix Reconfiguration Utility (KRU) automates the process of binding SDK-implemented components to Sammi's executable programs based on a configuration specification. Like the Class Definition Tool, the KRU consists of a macro-like configuration specification language and a compiler. The compiler processes configuration specifications to generate binding procedures and make files, then builds custom versions of Sammi and the Format Editor.

SDK programmers define a Sammi configuration in a KRU text file. The file contains statements that specify the command procedures, WAPs, DDOs, and user-implemented library files to bind into a new version of Sammi. This file is then processed by the KRU compiler to generate the specified configuration.

KRU configurations can inherit specifications from other KRU configurations in the same way derived classes inherit from their base classes. KRU inheritance can be effectively used to generate and manage multiple, specialized versions of Sammi for delivery to different groups of users or customers.

OPTIONAL SOFTWARE

DXF Converter

The Sammi DXF Converter is included with each Sammi Application Development Kit (ADK) purchased after January 21, 1998, or is available separately for a nominal charge. It is a batch-type program that reads AutoCAD™ and other CAD program DXF files and generates ASCII format files.* The generated format files can be manually edited (if necessary), then converted to binary format files for use in the Sammi Runtime Environment and Format Editor.

The DXF converter converts AutoCAD primitives to their nearest Sammi primitive. The supported primitives are points, lines, polylines, polygons, arcs, circles, ellipses, text, donuts, and 3d faces. In addition, the converter translates DXF composites and layers to Sammi composites and layers.

The program provides several useful options for controlling the DXF-to-Sammi conversion process. One option controls which layers are converted, another controls the dimensions of the generated format file, and another option controls whether all or a subset of the DXF primitives are converted.

* At this document's print date, the Kinesix' DXF Converter supports AutoCAD up to version 12.

Oracle Server

The Sammi Oracle server is used to connect Sammi DDOs to an Oracle database. These DDOs display values from the database, and write values to it through the Oracle server. This server is an API peer application that implements networked interaction between Sammi's runtime processes and an Oracle database. The Oracle server communicates with Sammi using Sammi's API protocol, and communicates with an Oracle database using the Oracle Call Interface. The Oracle server can reside on the same host CPU as Sammi's runtime processes, or on another CPU. Similarly, it can access an Oracle host on the same CPU, or on another CPU.

The Oracle server uses SQL statements to query and update the Oracle database. These SQL statements are placed in the read key and write key fields of the relevant DDOs. The Oracle server uses the read key information to query the Oracle database for a DDO's values, then sends the values to the Sammi runtime for display. When new values are entered into a DDO, the Oracle server uses the write key information to update the appropriate database record(s).

ABOUT KINESIX

Based in Houston, Texas, Kinesix Software develops, markets and supports software development tools used to rapidly deploy client/server and Web-based visualization & control systems with real-time, interactive graphics.

Our core product is Sammi®, a commercial off-the-shelf development toolkit with an advanced, multi-process architecture ideal for real-time command & control systems and mission-critical applications demanding reliability and high performance. It was voted "Product of the Year" in 1991 by UNIXWORLD, and was selected unanimously by the International Space Station Alpha Technical Committee over fourteen competing products in 1995.

Kinesix Software is the developer of Sammi, an enterprise and control-room graphics tool used by more than 20,000 mission-command and process-control workers in more than 40 countries. Kinesix also offers Sendera, a Web-based application that makes dynamic graphics viewable over the Internet to any Java-enabled Web browser.

With over 20,000 installations in 40 countries, Sammi provides engineers and IT departments with simple and efficient development, testing, and maintenance of graphical applications distributed throughout a network of mixed operating systems without writing any GUI or network code.

Kinesix also offers the Sendera™ Java Application Development Kit, a software application that extends live operational data from a control center environment to an enterprise via standard Web browsers. Operating within a three-tier architecture, Sendera manages the flow of information between back-end data sources and thin clients, giving every business unit in the enterprise the ability to monitor and control live data right at their desktops using Netscape Navigator, Microsoft Internet Explorer, or any other Java-enabled browser.

Whether for an international aerospace company controlling mission-critical space launch operations or a commercial enterprise monitoring process data from remote locations over the Internet, Kinesix helps organizations to develop and deploy intelligent systems for decision support and control. Our customers include NASA, Siemens, Invensys, Lockheed Martin, Bristol Babcock, General Electric, Boeing and IBM.

Kinesix is a full-service company providing product development, marketing, sales, technical support, consulting and training.