Report 84-34
Stanford -- KSL

Scientific DataLink

HELIOS User's Manual.
Gordon Foyster,
Aug 1984

card 1 of 1

Heuristic Programming Project
Report No. HPP 84-34

August 1984

HELIOS

User's Manual

GORDON FOYSTER
STANFORD UNIVERSITY
August 20, 1984

Contents

1. Introduction

The HELIOS system is a tool for conducting research into the design, test and diagnosis of computer systems. This manual describes how HELIOS can be used to capture and explore system designs.

Capturing a design means that the system must retain all the information about a design that the designer has given, sometimes only implicitly. This information can then be used by other parts of the system to refine the design so that it can be produced according to the designer's wishes. The system should make it as easy as possible for the designer to enter the design information, but also check that the information is correct according to its own knowledge of design. It should help the designer by providing alternatives to a design, for the designer to accept, modify or reject. The current system provides an easy to use interface to enter designs but provides little help with design consistency and correctness.

Exploring a design has several dimensions: Firstly we can simply view the system under design in space or through a component hierarchy. The design must also be explored as to its speed performance, power consumption, testability and manufacturability. Lastly, we may have several alternatives to the design at different levels and we need to know the effects of the different choices at other levels.

There are two parts to a design. The first is the structure of the system - which components are connected to which other components and which components are sub-components of other components. Interactive graphics driven by command menus is the medium for capturing structure. Components are represented by **boxes** with sub-components contained within component boxes. Connections are represented by means of **lines** between **ports**, which are attached to components. In order to represent forking or joining of connections **contacts** are used to show the equivalence of up to four connections. The menu commands allow addition, deletion and moving of component boxes, lines, ports and contacts. A window system allows different views of circuits under design.

The second part of the design is the behavior of the components. Behavior is the way a component reacts to external changes. We have used the ports of components to represent the interface of components with their environment. Thus the behavior of a component can be expressed in terms of its ports and internal state alone, and nothing else in its environment. Connections can also have behavior, which will be in terms of the ports or contacts at the two ends.

2. Getting Started

This manual assumes some knowledge of the operating environment provided by the Symbolics 3600. You need to know how to move from one system on the 3600 to another, how to use the editor, mouse and menus, and basic Lisp programming. In order to save and load circuits created by HELIOS you will need to know how to use the (distributed) file system at your site.

HELIOS is loaded by first loading the defining systems file. The location of this systems file varies with the installation and you will need to use site specific file names to find it. It is a good idea to have the ZETALISP variables **ibase** and **base** set to 10. (decimal). Then run **(make-system 'helios)**. This will load all the necessary files and start the HELIOS process. To start editing type **(helios)** to the Lisp Listener or use the <SELECT> H keys. If at any time during operation the HELIOS process stops, go to a Lisp Listener and type **(helios)** to restart the process.

3. Component Libraries and Circuits

After the HELIOS system has been loaded and started the user is given a mostly gray screen and a small window for interaction with the system. When HELIOS is being used the space not taken by the interaction window can be used to look at components. Top level commands are selected by toggling the right mouse button in the window at the top of the screen or any space on the rest of the screen not being used to look at a components. These top level commands are in the form of a menu with a variety of choices about creating, editing and saving circuits and libraries.

3.1 Loading and Creating Libraries

A component library is a set of components that can be used in a circuit. The components in the library may be be built using other components, which also must be in the library. In order to use the system to create circuits there must be a component library loaded. At present only one library can be loaded at a time.

Component libraries exist as text files that can be accessed through the 3600's file system. That is, they can be on remote machines, like other files used by the system. A naming convention is that the libraries are stored on files given by *Name*.X, where *Name* is the name of the library. To load a library select Load Library from the top menu and give the name in response to the prompt. You will also be prompted for the file path for the library file, which will vary between installations.

Library creation is done by copying an existing library, which must be already loaded. If no other library exists you can copy the PROTOKERNEL library. This library contains a minimal set of basic prototypes and is provided with the system. To create a copy of a library you select Edit Library from the top level menu and then Change Library Name from the next menu. Enter the new name in response to the prompt.

3.2 Modifying Libraries

Library modification means adding, deleting or changing the prototypes in the library. All of these options are available from the sub-menu of Edit Library from the top level menu.

Prototypes are added by selecting Create Prototype from this menu. You are asked whether it is to be a brand new prototype, which requires knowing some of the implementation details, or a copy of an existing prototype, in which case a prompt is given for the desired prototype to copy.

Prototypes are deleted by selecting Delete Prototype and selecting the one to be deleted.

To change a prototype select Edit Prototype and go into editing mode as described below.

After making desired changes, select Save Library from the top menu.

3.3 Deleting Libraries

Since one and only one library can be loaded in HELIOS at a time, you can make the system forget about the current library by loading another library. If a library will never be used again then you can delete the file containing the library using the normal file system operations.

3.4 Loading and Creating Circuits

Although you can only have one library loaded, at times you may wish to be exploring more than one design. Circuits are loaded by selecting Load Circuit from the top level menu and created by selecting Create Circuit. New circuits must have different names from circuits already in the

system. If names conflict then you will be asked to resolve the conflict by overwriting the existing circuit or, in the case of circuit creation, renaming the new circuit, or the process can be aborted. When you select Create Circuit you are shown a view of the new circuit and put into editing mode.

3.5 Modifying Circuits

Select Edit Circuit from the top level menu and select the circuit to be modified from those known to the system. You are then put into edit mode for that circuit. After the desired changes have been made you can save the new circuit by selecting Save Circuit from the top menu and following the prompts.

3.6 Deleting Circuits

To make HELIOS forget about a circuit, for example, in order to recover some space, select Delete Circuit from the top menu. If any windows are currently being used to look at the selected circuit the system will ask you to confirm. Just like libraries, circuits are stored as normal text files and can be deleted when no longer needed.

4. Worlds

Whenever you wish to perform some action on a component from a library or a circuit the component must be first put into a *world*. The world is a book-keeping entity which rembers the state of HELIOS with respect to that component. The information maintained by the world includes a list of windows (viewers) currently looking at the component and the current HELIOS mode (Edit, Simulate, Diagnose etc.) for the component.

4.1 Viewing

When you have created a world for your component, usually by invoking the editor on the component, you get a single view of the component. This view can be altered or new views created by clicking the RIGHT mouse button on the viewer for the component. This button causes a menu of commands to pop up which allows you to increase or decrease the scale of the circuit (Zoom and Set/Reset Region), alter the part of the circuit being viewed (Scroll), create new views of the component (Open Window) and set the depth of the component hierarchy to be seen (Set Display Level). This menu of commands is the same, independent of the current HELIOS mode for the component.

4.2 Modes

The left button menu in a viewer is a list of commands which depends on the HELIOS mode (maintained on the world) for the component. For example, in Edit mode the left menu is a list of commands to perform structural editing of the component. Current modes are Edit, Simulate and Diagnose. This manual will describe the Editing and Simulation modes in more detail while the Diagnosis system is in another manual (DART). The left button menu should always contain a command to change the HELIOS mode for the component. Note that a mode change is maintained by the world and the effects of the mode change will be visible in all views of the component. This explicit maintenance of modes allows new sub-systems to be easily added to the HELIOS system.

5. Component Structure

5.1 Hierarchies

Components in HELIOS are composed from other components. The graphical representation of this is that sub-components are shown as boxes which are always contained within the box of which they are a part. No other components, which are not sub-componens should be contained, even partially, within the component's boundary. At the lowest level of the hierarchy the components do not contain other components. For efficiency some component types cannot contain other components when they are used in a circuit. These types are derivatives of the PRIMITIVE component of the PROTOKERNEL library, in contrast with derivatives of the COMPOSITE component, which allows sub-components.

A box can have ports on its boundary to allow connections to lines in the higher level box. The ports can contain sub-ports which allows translation of data types between signals transmitted by the wires attached to the top level and contained ports. Once again, for efficiency, some ports can contain other ports (derivatives of the WIDEPORT in the library) and others cannot. The graphical representation of wide ports is determined by the width attribute of the port.

5.2 Editing Structure

To perform structural editing click the LEFT mouse in any viewer associated with the component. This brings up a menu which allows you to change the structure of the circuit, or allows you to change the mode to Edit so that you get the Edit menu.

The commands on the menu allow you to add boxes, ports, lines and contacts. Components of all kinds can be deleted or moved. Some menu commands require you to select the menu item with the RIGHT button in order to select the component to apply the command to. For example, to add a port to a sub-component and not the top level circuit, first click LEFT then RIGHT over the Add Port item. This will allow you to select the box to which the port is to be added. The menu commands are self documenting and provide prompts to help the user. At any stage during the execution of a menu command you can use the <ABORT> key to return to the top level and the command will be forgotten.

The structure of the component can be modified at any visible level. You can add ports to sub-

components and then connect a line to the port in the top level component. Component bounding boxes can be modified at any level to allow space to fit in more components or to compact after deleting components. The structure editing commands are not particularly smart so that most operations require the user to give completely explicit instructions about everything.

The menu commands try to conform to a consistent style. **Most actions require confirmation** before they will be carried out. This allows you to backtrack to the last step. Generally you use the LEFT button to SELECT positions and components, the RIGHT to CONFIRM and the MIDDLE to backtrack, or finish. If you try to add a line between two ports that currently do not have lines you will get opportunities to perform all of these actions.

Editing takes place on a world grid so that all component boxes, ports and lines are constrained to lie at integer co-ordinates. The default circuit size, which creating a circuit produces, is very large so that the grid almost looks continuous. To get a better feel for the grid, reshape the bounding box of the circuit to make it much smaller, then you will see that added ports can only take a number of discrete positions along the side of the circuit's box. Because of the underlying world grid it is possible to align components exactly, allowing straight lines for wires and discrete measures to be used in component positions.

Components that have been created by means of the editor can be put into the library using the Prototize command from the edit menu. This also can be done for components (boxes, ports or lines) which have been taken from the library and modified in the circuit.

The Edit Behavior command is provided for convenience in the edit menu. It will be dealt with in the next section.

Other commands on this menu are useful when you know some of the implementation details. Inspect Component and Inspect World allow you to invoke the inspector on the internal structure of a selected component or world. Instantiate Box is required because of some efficiency issues which have not yet been made invisible.

6. Component Behavior

6.1 Behavior Language

The language used to specify behavior and allow simulation is based on the CORONA language (Singh, Genesereth). We are using a subset of the language so that we can easily compile the entered form of the behavior into procedures to make simulation faster.

Behavior for components is entered, at the top level, by using a set of *implication rules*. These rules may use attached procedures, written in LISP. The rules are used to specify the *simulation events* and *timing*.

For example: (if (TRUE *event1 time1*) (TRUE *event2 time2*))

Where TRUE is a keyword used to denote an event happening at a particular time.

An *event* is one of (Val *port value*) or (Val *state value*)

Val is a keyword to denote an event and allows attached procedures to perform specialised storage and retrieval operations.

A *port* is specified by (Port *name*) or a *variable* or a function that returns a port instance.

A *state* is a way to reference some attribute of the component for which we are entering the behavior. This attribute is normally an instance variable of the flavor for the component. It is specified by (State *name*). Where *name* is the instance variable name. The implementation section describes how components are flavor instances.

Value may be a constant, a variable or a function.

Time is either a variable or a function returning a number. Usually time will be a variable (that will be bound to the current simulation time during simulation) on the LHS of rules and a non-decreasing function of that time on the RHS of rules. The correspondence between time in rules and real time is defined implicitly by the writer of behavior rules.

A *variable* is any symbol beginning with $. Conceptually, variables are bound to values. During simulation all variables on the LHS of behavioral rules will be bound if the LHS is satisfied. These bindings of the variables can be used in functions on the RHS of rules, or in clauses following the binding clause on the LHS.

The special keyword SELF is used to refer to the object whose behavior is being specified. This variable can be passed to functions that are referenced in the behavior rules.

Functions may reference variables, providing they have been bound earlier in the rule, or constants. Functions can also return values through unbound variables which must be bound to a value in the function, ie. the function must return an *assoc* list.

Both the LHS and RHS of the implication rule may be conjuncts (AND). The conjuncts may be of temporal rules or other predicates. Other predicates are assumed to be defined as LISP functions by the rule compiler. The LHS rule may contain one disjunct (OR) at the top level. That is, the LHS can be a disjunct of conjuncts.

Some rule examples:

```
(if (and (TRUE (Val (Port IN) $S) $T)
         (InSignal $S)
         (Val (State mystate) $CS))

    (and (TRUE (Val (Port OUT) (OutFn $S)) (Delay $T))
         (TRUE (Val (State mystate) (NextState $S $CS)) (ADD1 $T))))


(if (and (TRUE (Val $PORT $S) $T)
         (Destport $PORT $S))

    (and (TRUE (Val (Port OUT) $S) (ADD1 $T))
         (TRUE (Val (State myslot) (NewSlot $PORT $S $T)) $T)))
```

The next two behavioral rules give the behavior for an AND gate:

```
(if (AND (TRUE (Val (Port IN1) 1) $T)
         (TRUE (Val (Port IN2) 1) $T))
    (TRUE (Val (Port OUT) 1) (ADD1 $T))


(if (OR (TRUE (Val (Port IN1) 0) $T)
        (TRUE (Val (Port IN2) 0) $T))
   (TRUE (Val (Port OUT) 0) (ADD1 $T))
```

For a port with 4 subports the behavior could be:

```
(if (TRUE (Val SELF $V) $T)
    (AND (TRUE (Val (Port 1) (Bit-1 $V)) $T)
         (TRUE (Val (Port 2) (Bit-2 $V)) $T)
         (TRUE (Val (Port 3) (Bit-3 $V)) $T)
         (TRUE (Val (Port 4) (Bit-4 $V)) $T)))
```

```
(if (AND (TRUE (Val (Port 1) $V1) $T)
         (TRUE (Val (Port 2) $V2) $T)
         (TRUE (Val (Port 3) $V3) $T)
         (TRUE (Val (Port 4) $V4) $T))
    (TRUE (Val SELF (+ (* 8 $V4) (* 4 $V3) (* 2 $V2) $V1)) $T))
```

For a component with many ports having similar functions it is often convenient to use a variable for a port. This can be done in two ways, for example:

(TRUE (Val $PORT $V) $T) will bind $PORT to the port **instance** in any port event for the component. Use of this binding requires knowledge of the IMPLEMENTATION section.

(TRUE (Val (Port $PNAME) $V) $T) will bind $PNAME to the port name for any port event for the component.

A very useful predicate function to be used on the LHS of rules is the = function. This function takes two arguments. If both arguments are unbound variables the predicate is true. If either of the arguments is an unbound variable it is bound to the other non-variable argument. This other argument can be one of a constant, or a bound variable, which will give the unbound variable the same binding, or a function, which will bind the unbound variable to the result of evaluating the function. If neither of the arguments are unbound variables the function returns T if they are equal. For example, an AND gate behavior could use the logand function to specify its behavior as follows:

```
(if (AND (TRUE (Val (Port IN1) $V1) $T)
         (TRUE (Val (Port IN2) $V2) $T)
         (= $RES (logand $V1 $V2)))
    (TRUE (Val (Port OUT) $RES) (1+ $T)))
```

6.1.1 Restrictions

Some of the restrictions on behavior rules imposed by the rule compiler:

Variables can NOT be used for state names, ie you cannot say
(TRUE (Val (State $S) $V) $T or (TRUE (Val $STATE $V) $T)

Functions which are used to return binding lists can currently only accept a single unbound variable and return a binding list with only the binding of that variable or NIL.

6.2 Editing Behavior

Most of the commands dealing with behavior are invoked from the LEFT button menu when in Simulate mode. The behavior of any component can be changed by selecting Edit Behavior and then the component. The behavioral rules for the component will be inserted into a ZMACS buffer to be edited. When the component is new you will be given a template for a single behavioral rule, according to the above syntax. You can have any number of rules in your set for a device. You exit from the buffer with CTRL-Z and have a choice of using the new rules or the old.

If behavior needs to be changed for all instances of a prototype then the behavior of the prototype in the library should be edited. If the behavior if a component in the circuit is edited, it will be forever disconnected from the behavior of its prototype in the library.

6.3 Simulation

To simulate change the mode to Simulate using the LEFT button menu. In Simulation mode

the LEFT menu is changed to give the simulation commands.

The initial conditions for simulation are called Test (for test vectors) on the command menu. The test vectors can be loaded from, and saved to files, deleted and modified. Modify Test allows tests which are initial values for ports to be changed. Initial values can be set to occur at any time during the simulation.

To see the ongoing simulation results, select Add Text Traces and the ports which you want to trace.

When simulating a component the simulator has to decide whether to use the behavior of the component as a whole (top behavior) or the behavior given by the interconnected sub-components, if any (internal behavior), or both of these behaviors concurrently. The Set Simulation Level command allows you to specify which behavior level is desired. You should enter the desired levels as a list of NIL and T according to the prompt.

The Simulate command allows you to go without stopping (No Break) or set break times (Break Step, Break Time) or just do the events at the next time (Single Step). Using the ABORT key during simulation causes a break from which the simualtion can be restarted. When the simulator is reset a function called User:UserSimResetFn is called with the circuit as a parameter. With some knowledge of the implementation this allows the user to write a function which will reset state slots of components.

The simulator uses compiled forms of behavior rules for rapid simulation. This allows you to reference the simulated component inside functions called from behavior rules. In particular it allows you to use the defun-method construct for these functions. (See the ApplyMicrocode function with the IBMPC example.) To take a look at the Lisp source form of the last set of compiled behavior rules evaluate the global variable User:LatestBehavDefn.

6.3.1 Control

By invoking the Simulator Control command on the simulation menu you can change some important aspects of the way the simulator works or produce side effects while the simulation is proceeding. Possible side effects include the production of animated graphics to show the simulation progress or recording of simulation data.

By invoking the Simulator Control option with the RIGHT mouse button you can edit a set of rules which have the same syntax as the behavior rules. These rules are attached to the simulator and are checked against every simulation event. You will probably want to invoke LISP procedures from the RHS of the simulator rules, rather than creating new simulation events.

The LEFT mouse button on the Simulator Control option causes a menu to appear with some options to control the action of the simulator internals. These options are currently:

Allow zero delay components - If set to YES then the delays of box components can be zero. This setting allows some errors with propogation of signals when all the inputs have not been set and zero delay feedback loops, but is generally more useful than the problems caused.

Don't flag inconsistent signals - If set to YES then the simulation events are not checked to see that the same state or port is not set to different values at the same time. If NO then an ERROR assertion is made (currently put into the simulation event queue) giving the place and time of the inconsistency. This ERROR assertion could be trapped by the simulator rules.

Propagate inconsistent signals - If YES then all signals are propagated, whether the same port or state has been set to different values at the same time. This parameter acts independently of the inconsistent signal error flag parameter. If set to NO then new port or state values for the same time will over-write old values. This is a useful feature when a component receives many simulataneous input signals which can affect a single output and you only want the effect of the last input to be propagated. A particular application of this is in the behavior of a bus when you want to propagate the incoming signal on a bus via a WIDEPORT component to its sub-ports so that the bus signal is split up into bit signals. This parameter should probably only be set to NO when Allow zero delay components is also NO.

All these parameters default to YES, which produces the fastest and generally most useful mode.

6.3.2 Instrumentation

An interface exists that allows the user to customize the display of simulation results. Port values and state changes can be monitored by means of *instruments* which the user attaches to components. One instrument may monitor many port or state changes in order to collect the information and display it in a summary form. Values can be monitored in two ways:

1. Triggered. Whenever the selected port or state changes the instrument receives a signal describing the change. A special kind of trigger event is the change in the simulator event time. An instrument can use this trigger event to update its display of information.

2. Polled. The instrument updates its knowledge of a port or state value at a rate specific to that instrument and port or state. Thus if the user knows that a particular value changes slowly it is polled at a lower rate while the same instrument might poll other values faster.

The following paragraphs contain some implementation information which describes instrument interaction with the simulator via messages. This assumes that instruments will be instances of FLAVORS and that particular instrument instances will receive messages sent by the interface to the simulator.

A new instrument is created by selecting New Instrument from the simulation menu. A menu of known instrument types (flavors) is then presented. When the instrument type has been selected a new instrument of that type is created and the new instrument instance is sent a Position message with the viewer containing the circuit as a parameter. The instrument handles this message by positioning itself, possibly with user interaction, with respect to the viewer.

Instruments can be used by selecting Connect Instrument from the simulation menu then the particular instrument and where (port or state) it is to be connected to. If the value is to be polled then the polling rate must be specified. If changes in the value are meant to trigger the instrument

then you specify either that any changes will cause a trigger or that changes to a particular value cause a trigger. When a new connection is made from an instrument to the circuit the instrument is sent a New-Connection message with the connection object and state being monitored by that connection. Each time the interface between the simulator and instruments detects a change in a monitored value it sends a New-Value message to the monitoring instrument with the object, state and new value as parameters.

At the moment the only instruments are individual displays of port or contact values or a moving window on a collection of port values.

7. Menu Summaries

This section summarises the commands in the four main menus of HELIOS.

A. Top Level Menu

Load Circuit. Prompts the user for a circuit name to load from file.
Load Library. Load a prototype library from file.
Create Circuit. Create a new circuit.
Edit Circuit. Edit a circuit which was either loaded from file or created earlier during the current session.
Edit Library. Edit the current library. A submenu allows creation, modification and deletion of prototypes or changing the library name so that the library will be a copy of the loaded library.
Save Circuit. Will save a selected circuit to file.
Delete Circuit. Makes HELIOS forget about a selected circuit.
Save Library Saves the current library to file.

The following three menus are invoked by mouse clicks in a window containing a view of a HELIOS component.

B. Structure Edit Menu

Add Box. Add a box from the prototype library to the component. Invoking this command with the RIGHT button allows selection of the sub-component which will contain the new box.
Add Lines. Add a number of lines between unused ports or contacts.
Add Ports. Add new ports to either the top box or a selected box, if the RIGHT button is used.
Add Contacts. Add new contacts. It is easy to make a mistake with this command as HELIOS currently does not check that the contact is in a legal position. This means that a contact can be added to the top-level box so that is looks like it is inside a sub-box. The contact then cannot be deleted without moving or deleting the sub-box. Remember to use the RIGHT button when trying to add a contact to a sub-box.
Move Components. Ports, contacts and boxes can be moved when they have no attached wires.
Delete Components. Any kind of component can be deleted.
Reshape Bounding Box. The bounding box of boxes without ports can be reshaped. This constraint on not having ports encourages a bottom-up style of construction.

Edit Behavior. Is in this menu for convenience. It allows the behavior of a selected component to be changed.
Modify Attributes. Allows miscellaneous attributes of components to be changed

Instantiate Box. This particularly applies to ARRAY components, when the user wants to see the replicated cells in the array. This command has effects which cannot easily be undone so should not be used until necessary.
Prototize Component. Allows a box to be taken from the component being edited and put in the prototype library.
Inspect Component. Invokes the Symbolics Inspector on the data structure for a selected component.

Inspect World. Invokes the Inspector on the world containing the component being edited.

CHANGE MODE. Change the HELIOS mode for this world.

C. Simulation Operations

Simulate. Start a simulation. A sub-menu allows setting of the break and step conditions.
Reset Simulation. Reset all state to allow a new simulation.

Load Test. Load a set of test vectors for simulation from file.
Save Test. Save the current set of test vectors to file.
Flush Test. Delete the current set of test vectors.
Modify Test. Add new test vectors of modify existing test vectors. New vectors are added by selecting the port to be forced to a value, the value and the time when the value is to be set.
Modify Attributes. The same as in the Edit Menu, provided here for modification of simulation-like attributes.

Edit Behavior. Edit the behavior of a selected component.
Simulator Control. Allows checking and changing of the three switches controlling the simulator, as described above.
Set Simulation Level. Allows the simulation level for a component to be changed. In most cases the default simulation level is correct.

Inspect World. Inspect the world containing the component.
CHANGE MODE. Change the HELIOS mode for the world.

D. Window Operations

Redraw Window. Redraws the contents of the window.
Reshape Window. To make the window on the screen larger, smaller or a different shape.
Move Window. Move the window to a different place on the screen.
Rename Window. Put a new name in the bottom of the window.
Bury Window. Put window at bottom of overlapping HELIOS windows on the screen.

Set Display Level. Set the level in the component hierarchy to be displayed.
Set Region. Select the part of the component to be displayed by the window.
Reset Region. Displays all the component in the window.
Zoom In. Display less of the component but with greater detail.
Zoom Out. Display more of the component with less detail.
Scroll Region. Display different part of the component but with same scaling. That is, move the window to show a different part of the component.
Open Window. This is a command to tell the world that we need another window on the same component.
Close Window. Close this window on the component.

IMPLEMENTATION

HELIOS evolved from several sources, in particular the Boxes and Lines editor (BAL) of Terry Barnes. Some of the names used in HELIOS reflect this history.

1. Flavors

All prototype components in the system are instances of flavors. The system provides two basic flavors - PRIMITIVE-BOX and COMPOSITE-BOX. PRIMITIVE-BOX instances can not be decomposed in a component hierarchy into sub-components - they have no slots for referring to sub-components. COMPOSITE-BOX components can have three types of sub-components - BOXES a list of component instances, LINES a list of instances used to interconnect the boxes and CONTACTS which join up to 4 lines together. The user may wish to specialise either PRIMITIVE-BOX or COMPOSITE-BOX to add slots not present on these flavors. All graphical editing of instances of the specialised flavors will still work because of message inheritance. (See the IBMPC example to see use of specialisations). Components which are instances of the specialised flavors can be added to the current library by selecting EditLibrary at the top menu then CreatePrototype then typing the name of the flavor of which the new library component will be an instance.

PORTS, LINES and CONTACTS can also be instances of specialised flavors. The basic flavors are BAL-PORT, BAL-LINE and BAL-CONTACT. The port flavor has a specialisation in the system called WIDEPORT which allows the user to add sub-ports to ports. Wide ports can have behavior which describes how signals arriving on sub-ports affect the signal on the wide port and vice versa. The line flavor has also been specialised by mixing in the COMPOSITE-BOX flavor to create COMPOSITE-LINE. Instances of this flavor can have sub-components like composite boxes but look like lines when viewed at their top level. When viewed at lower levels these composite lines look like composite boxes with a dashed bounding-box.

At the moment the mechanism for introducing specialised instances of non-box components into the library is a bit more crude than for boxes. First you need to load the library into HELIOS. Then go to Lisp and create the required instance using the system function **(make-instance** *flavor-name . . .***)**. This new instance should be given a name either in the make-instance or by using (send *new-instance* ':set-name *name*). Then to add the flavor to the library execute:

(send (send B:EditorOb ':LibraryIndex) ':Add *new-instance*)

After this the new instance will appear on the menu of prototypes when you add one of its type into a circuit.

2. Packages and Variables

Most of the system is in package BAL. The HELIOS system defines the name B to reference the BAL package. B:BOX-LIST is a list of created circuits and B:WORLD-LIST is a list of the worlds containing these circuits for editing.

Some parts of the system, developed independently from the main structural editing part, are in the USER package. This includes the reasoning system MRS, most of the simulator, MARS, and the mechanism for maintaining a textual representation of instances on file.

3. Structure

Invoking the inspector while editing a circuit is a good way to find out the internal representation of things and is also a useful way to perform structural modifications behind the

system's back. I would like to know of any instances when you need to do this as it indicates something lacking in the system.

When you add an instance of COMPOSITE-BOX to the circuit you actually add a descriptor component which points back to the library component which is it's prototype. In order to edit or simulate this component in the circuit it first has to be **instantiated**. That is, the descriptor must be replaced by a real instance of the object with all the internal components in place. The Instantiate Box command on the edit menu does this for you. The reason for having this indirection is to save space in representing circuits with many levels of hierarchy. We do not need to reproduce many copies of the same sub-component information. In most cases the system knows when a component must be instantiated but in some cases the user must do it explicitly. The usual way to find out when a component needs to be instantiated is that the code will break with an error which indicates that an instance of the **BOX-DESCRIPTOR** flavor has received a message which it didn't understand. In this case just resume HELIOS and explicitly instantiate the component in question.

A hook is provided to allow you to attach specialised updating routines to your devices. This is done through the ModifyAttributes command on the left button menu. When this command is selected the system asks for a device, and if the device has a handler, it sends a :ModifyAttributes message. You can define your own method to handle the message for your devices.

4. Printing

Components are stored in text form in files by using the **file-print-mixin** flavor as a mixin to all flavors that need to have instances saved. You can therefore edit these files if you need. As part of the reading of instances from file every instance needs a unique identifier, supplied as slot UID by file-print-mixin. This UID is generated from the real time clock and a counter. A hash array called FLAVOR-OBJECT-ARRAY, keyed by the UID, contains all instances read from file.

In order to save space on printing and time on loading it is possible to prevent selected instance variables of objects from being printed out. These instance variables will be initialised to their default value as given by the flavor when the object is read back in. If you have your own specialised flavors for devices and have some instance variables which are only used for book-keeping during execution of some program, you may not want the values of these variables to be saved when the device is saved to file. To prevent instance variables from being saved to file define an :After method for the flavor:

(Flavor-name :After :AddNonPrintIVs)

This method should simply push the name of the instance variables not to be printed onto the instance variable User:Non-print-ivs

For example, if you don't want to have instance variable "foo" printed:

```
(defmethod (MyFlavor :After :AddNonPrintIVs) ( )
    (Push 'foo User:Non-print-ivs))
```

After an instance has been loaded from file (via the DEFINST function) a :LoadedFromFile message is sent to it. This facility allows you to specialise the behavior of an object after loading. It is used by the system to compile the text form of behavior rules to give a compiled lisp routine for simulation, since this compiled form is not written to the file.

5. Specialised Simulation

5.1 Procedures for Behavior

There are two ways in which you can write your own procedures to be used instead of rules for simulation. This does not include the attachment of function calls from the behavioral rules, but writing a function which completely performs the desired behavior.

1. Changing the Compiled Form.

Any device to be simulated will have an instance variable called :SpecifiedBehavior. This

slot points to an instance of the B:BehaviorRules flavor. This instance will have a slot called :CompiledForm which is the actual compiled function which will be called when the original device is to be simulated. You can use the inspector to change this slot to be the name of your own function. This function must take the name of the device as the first parameter and the simulation event as the second parameter.

2. Change the Method

The simulator works by sending a message to a device which has an event at the current simulation time for that device. The message is :ApplyRules and the event is a parameter. This message is normally handled by the B:BehaviorObject mixin. If you have your own specialised flavors which are being instantiated for devices then you can define your own :ApplyRules method for those flavors, to avoid going through the indirection of the BehaviorRules instance.

In both the above cases the function will receive be passed the simulation event which might causes some action. This event is a list of the form:

(TRUE (Value *object slot value*) *time*)

Where *object* is the object to which the event happened.
slot is the affected instance variable of the object.
value is the new value of the slot, and
time is the time of the event.

When the event is something happening to a port the the object will be the port and the slot will be :Signal. A utility function (GetNamedObject *owner object-type name*) is provided to retrieve sub-parts of objects. For example to retrieve FOO's port named "IN" use (GetNamedObject FOO 'Ports 'IN).

To produce further simulation events at a new time, call the function:

(User:Make-Sim-Assertion *object slot value new-time*)

The global variable User:LatestBehavDefn contains the source code of the last set of compiled behavior rules, if you want to see the form of code to be used for simulation.

NOTE: The above description of procedure definition also applies to the rules for the behavior of the simulator. At every simulation step the simulator's behavior is called for every event which happened at that step. This allows you to produce graphics effects or record information for that step.

5.2 Simulation Reset

Besides making your own functions for simulation you might want to change the effects of resetting the simulator. There are again two ways:

1. User:UserSimResetFn is called every time the simulator is reset with the circuit as a parameter. You can redefine this function to do your own thing. The function provided with the system does nothing.

2. Again, if you have your own specialised flavors for devices you can define a method :SimReset for the device which will be called when the simulator is reset. However, in this case the default :SimReset method provided by the system does something useful, so your method should be a :Before or :After type.

6. MRS

A set of routines exists so that MRS can use the underlying representation, based on flavors, to perform reasoning. This set of routines produces a propositional representation according to the SUBTLE manual. Currently they are only for looking-up or accessing the design data and are accesed through the single routine **obj-lookups** which then dispatches to perform the particular lookup. The file "HELIOSMRS" contains the lookup routines.

7. Menus

All of the menus used by HELIOS and most of the global variables are defined in the file "VARS". The top level menu is attached directly to the gray background window which handles mouse clicks in that window. The menus for performing the current mode commands for a world and the viewport-altering commands are attached to the world of the viewport. When a mouse is clicked in a viewer the viewer first checks to see if it can handle the mouse button. When it can't - the usual case - it looks on its world for the Button-Table of the world. The viewer then passes the routine from the buton table, as given by the clicked button, to the B:BAL-PROCESS. The routine is then evaluated by the B:BAL-PROCESS - usually resulting in a menu being produced. By passing the routine off to the BAL-PROCESS we avoid tying up the mouse process to do the resulting computation. The button tables for different world modes and the definitions for the top level BAL process are in the file "MAIN".

8. Distributed Simulation

A new mode has been added to HELIOS to allow simulations to be distributed over multiple 3600s connected by a CHAOS net. The user interface to use this mode is as follows:

On each of the machines to be used the user must start up HELIOS and load the circuit to be simulated. That is, the full circuit must be loaded on all machines, you cannot have the circuit split into smaller parts which will be simulated independently. Then, on each of the machines, you enter the DISTRIB SIM mode for that circuit. The command menu in this mode is the same as for normal simulation except for the Partition Circuit command.

When each machine is in DISTRIB SIM mode you should have in mind some mapping of the component boxes of the circuit onto the available machines. This mapping should cause the smallest amount of communication between machines and, if possible, an even workload on the machines. This mapping must be entered by the user and connot be re-arranged during the simulation run. With this mapping in mind you should load or modify data for the components being simulated, on the machines they will be simulated by.

Now, select the machine which will be the controller of the other machines during the simulation. On this machine select the Partition Circuit command. You will then enter a loop where you are first asked for the name of a server machine then asked to select those components that machine will be simulate. You should be careful not to say that two machines will simulate the same component. Any components not selected will be simulated by the controller itself. The loop is terminated by typing ( ) in reponse to the request for the next machine name. HELIOS then changes the behavior of all wires which pass between components not simulated by the same machine so that these wires will now pass CHAOS packets between the machines, with the value of the signal along the wire.

You start and stop the simulation in the normal way, by using the simulation commands from the controller.

Implementation Issues:

To produce the distributed simulation a DistribSim flavor was created as a specialisation of the Simulator flavor. This new flavor should do normal (non-distributed) simulations correctly but there is some overhead in time. If this overhead is too great or bugs are appearing in normal simulation the special variable SimulatorFlavor should be changed back to 'Simulator instead of 'DistribSim. This variable is set in the VARS file and needs to be set before circuits are loaded.

When DISTRIB SIM mode is entered on any machine a new process is created which will listen for commands from the controller. When you have selected the controller by using the Partition Circuit command this process is killed on the controller machine. After the controller announces itself the server machines starts temporary processes which listen for messages to change the behavior of a number of wires. A new CHAOS connection is made between those machines which have interconnecting wires in the circuit.

During simulation the controller decides when each simulation step is started and stopped. To do this it processes its own simulation events then polls the other machines to find out if everyone is done. When they all are done it moves on to the next step. In order to avoid a machine thinking it is done while it still has to receive and process a signal on a wire from another machine, all machines send out DONE messages along the connecting wires when they have no more messages to send. Now any machine cannot tell the controller it is finished the current step until it receives DONE messages on all external wires. This scheme can still fail but should be robust enough for most situations.

Because of the overhead in using the CHAOS net and the many messages to find the end of each step, the distributed simulation mode may be up to 10 times slower than normal simulation, if the circuit is unsuitable. To make use of this mode you need a minimum of communication between components simulated on different machines, and a large number of events taking place for each time step, for each of the parts.