HP Instrument Driver Language Reference

HEWLETT PACKARD

HP Part No. E2001-90004
Printed in USA

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated into another language without the prior written consent of Hewlett-Packard Company.

©Copyright 1989, 1992 Hewlett-Packard Co.

Printing History

First Edition - July 1992

Contents

1. The HP Instrument Driver

2. The Component Section

5. Writing a Driver

Figures

Tables

1

The HP Instrument Driver

Overview

The purpose of this manual is to provide a guide for writing a Hewlett-Packard Instrument Driver (HP ID). An HP ID is a collection of statements from the HP Instrument Driver Language that displays the instrument panels and controls the instrument over an interface bus (HP-IB, VXI, etc.) The HP ID allows the user to control the instrument both interactively (through the soft front panel) and programmatically. This language will be discussed in detail throughout this manual. An HP ID may include an optional Help file that can be read or printed by the user of the HP ID.

The HP Instrument Driver Language has proven powerful enough to describe a wide variety of programmable instruments.

This manual assumes familiarity with the use of HP Instrument Drivers.

Note

This book is written using *HP ITG* syntax. Instrument driver developers are responsible for knowing the language and limitations of their own development environment.

Conventions

The following typefaces are used to help you understand how terms and phrases are used:

Bold **Bold** indicates the introduction of new terms.

Italics Words are printed in *italics* for emphasis.

`Computer` Commands you should type or that appear in program listings appear in a computer-style type.

Key Individual keyboard keys are distinguished by the keycap-style border around the key's name.

Controlling an Instrument (Overview)

Any instrument, such as a voltmeter, function generator, or oscilloscope, must include some type of **interface** to be useful. This interface provides a set of controls that are used to configure the instrument to produce the desired result. Every time a control is changed, the operation of the instrument is changed to reflect the new setting. The controls provided by the interface allow the operator to change the state of the instrument. Some examples of controls for instruments are the frequency control of a function generator, the voltage setting of a power supply, and the time scale of an oscilloscope.

One type of interface is the front panels manual interface. The front panel allows the operator to change instrument settings manually to produce the desired state. The manual interface, however, is slow and does not allow automation. Every time a new instrument state is desired, the settings of each control must be changed manually, one at a time.

Because manual operation is error prone and tedious, many instruments provide another type of interface that allows controls to be set by computer commands. This type of interface is known as an interface bus. An interface bus allows many different devices to talk to each other. There are many different protocols that allow communication between devices, such as RS-232 or IEEE 488.2 (HP-IB); each one sets different standards for communication to take place. These standards ensure that devices know when to talk and when to listen, but most do not define a common language for all devices. As a result, instruments typically speak their own language, which means that a computer must be able to talk to each instrument separately. For example, to query the frequency on an HP3314 function generator, send “QFR”, but to query and HP3325 function generator, send “FR?” With HP-IB, each instrument is assigned to a different address and the controller must select the proper address to send a command to the instrument.

The interface bus allows a computer to change the settings of an instrument quickly—much quicker than the manual front panel interface allows. However, to write a program to change these settings, the programmer must have a knowledge of the language the instrument uses, that is, the programmer must know what commands to send to the instrument to change the correct control to the proper setting. By controlling the instrument over the interface bus, settings can be quickly changed and states quickly recovered. This set of commands, known as the **command set**, has syntactical and other rules that the programmer must adhere to.

The HP ID combines the speed, accuracy, and automation of the interface bus and the convenience of the front panel interface. Using an HP ID, the operator no longer needs to be concerned with the command set syntax of the instrument, instead, the HP ID knows how to communicate with the instruments it is controlling. The control information is provided in the form of a file called an instrument driver.

Note

If you’re serious about writing an HP ID, it will help to have an HP 59401A Bus System Analyzer or equivalent on hand to debug HP IB transactions generated by the driver.

Parts of an HP ID

This section describes in detail the different parts of an HP ID.

All HP IDs consist of two parts: a component section and a panel section. The component section describes how the HP ID will communicate with the instrument; the panel section describes how the HP ID will communicate with the user through the soft front panel, the user interface for the HP ID.

A minimum HP ID *must* include:

- A REVISION statement at the top of the HP ID.
- At least one component declaration.
- A main panel. Panels nest inside one another, therefore, any additional panels must be included as subpanels under the main panel.

The REVISION statement indicates which version of HP ID Language was used to write the HP ID. For instance, HP IDs that use the SYNC feature must announce themselves as requiring REVISION 2.0, since the SYNC feature was unavailable before 2.0. Currently, all HP IDs should be written with REVISION 2.0.

Component Section

The component section of an HP ID is a series of component declarations. A component is a placeholder for a value of a given control in the instrument. A component is similar to a variable in a programming language like BASIC or C. For example, suppose that you are writing an HP ID for a multimeter and want to allow the user of the HP ID to change the function and take a reading; declare two components, function and reading. First function:

```
COMPONENT Function;
  TYPE DISCRETE;
  VALUES DCV, ACV, OHM;
  INITIAL DCV;
END COMPONENT;
```

This component declaration can be thought of as a declaration of the variable “function”. (Case is not important.) “TYPE DISCRETE” means that the value for “function” must come from a list of values, here DCV, ACV, and OHM. When the multimeter is reset, function will revert to its initial value, DCV.

The component declaration for reading is

```
COMPONENT Reading;
  TYPE CONTINUOUS;
END COMPONENT;
```

“TYPE CONTINUOUS” says that this value can be any real number.

So far, these components are just variable declarations, as in BASIC; they only hold a value. If you want the user to be able to change function or take a reading, add SET and GET ACTIONS inside the component declarations. SET ACTIONS are executed when the user wishes to change the value of the component. GET ACTIONS are executed when the user wishes to ask the instrument for a reading.

Here is our completed function component:

```
COMPONENT Function;
 TYPE DISCRETE;
 VALUES DCV, ACV, OHM;
 INITIAL DCV;
 SET ACTIONS:
  OUTPUT Function TABLE "FN0", "FN1", "FN2";
 END ACTIONS;
 GET ACTIONS;
  OUTPUT STRING "FN?";
  ENTER Function FORMAT 'K';
 END ACTIONS;
END COMPONENT;
```

Now, when the user of this HP ID changes the function to something like ACV, the value of function gets changed to ACV, then the SET ACTIONS are executed. The OUTPUT TABLE statement causes one string to be sent to the instrument. If the value of function is DCV (the first value in the VALUES statement), then “FN0” (the first string in the OUTPUT TABLE statement) is sent. If the value of function is ACV, then “FN1” would be sent, and so on.

If the user wants to ask the instrument “What is the current function?”, then the GET ACTIONS are executed. This sends “FN?” to the instrument, which responds with a 0, 1, or 2, interpreted as DCV, ACV, and OHM, respectively.

Each component of the HP ID corresponds to a different control on the instrument, usually on a one to one basis. Order of the components in the file is important only for state recall, which will be covered later. The most often used components should be at the top since components are searched from top to bottom. That is why “READING” is usually the first component in a multimeter or counter HP ID.

Each component will have a name, type, and initial value, and most will have SET or GET ACTIONS, as well.

Panel Section

The panel section of the HP ID describes the soft front panel, the user interface for the HP ID. Usually, the panel section is easy to write and follows the menu structure of the instrument's HP-IB command set. That is, the panel section will include a subpanel for every different menu path on the HP-IB command set. Although it is possible to write an HP ID that does not reflect the HP-IB command set menu structure, there are several good reasons to follow the structure:

- The overhead is much less. When an HP ID is written with a different menu structure than the instrument has, extra code is almost always involved.
- For users of the HP ID that are already familiar with the HP-IB operation of the instrument, the panels will be easier to use.
- Very often, the HP-IB interface to the instrument is designed with a similar menu structure to that of the front panel interface. If this is the case, the panels of the HP ID will end up appearing and behaving similarly to the front panel of the instrument.
- Instrument menu structures are not developed at random. They are developed to fit the functionality of the instrument. Writing an HP ID to follow the menu structure preserves this functionality.

Panels are nested one inside another and the outside panel, usually called root or main, must appear after the last component in the HP ID. Each panel contains one or more “widgets”, or controls, that correspond to components in the HP ID. The type of each control is associated with the type of each component in the HP ID. Each “widget” controls one component from the component section. Occasionally, one component will have more than one widget associated with it. For example, an autoranging multimeter may have a discrete range control and an autorange on/off control.

As the user of the HP ID clicks on the widgets in the panel section, the HP ID executes the SET or GET action statements for the component tied to that widget.

HP ID Example

The following example illustrates the basics of building an HP ID.

In our example code, we have created a simple HP ID that lets the user control the function and range controls on the multimeter, then take a reading. The multimeter needs three components: function, range, and reading. As you SET the values of these components, the HP ID sends commands over the HP-IB to ensure that the multimeter is in the same state as the HP ID. It is also possible to query the instrument as to its current function, range, or reading. Querying is done by executing GET ACTIONS.

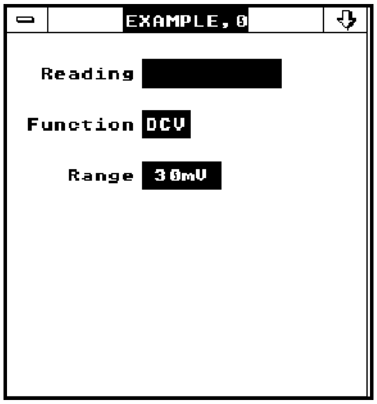
The panel section places all three components on the main panel.

Here is a simple HP ID for the multimeter. Figure 1-1 shows the panel it produces.

```
REVISION 2.0;

COMPONENT Reading;
        TYPE CONTINUOUS;
        GET ACTIONS;
                TRIGGER;
                ENTER Reading FORMAT 'K';
        END ACTIONS;
END COMPONENT;

COMPONENT Function;
        TYPE DISCRETE;
        VALUES DCV,ACV,OHM;
        INITIAL DCV;
        SET ACTIONS;
                OUTPUT Function TABLE "FN0","FN1","FN2";
        END ACTIONS;
        GET ACTIONS;
                OUTPUT STRING "FN?";
                ENTER Function FORMAT 'K';
        END ACTIONS;
END COMPONENT;

COMPONENT Range;
        TYPE DISCRETE;
        VALUES "30mV", "300mV", "3V", "30V", "300V";
        INITIAL "3V";
        SET ACTIONS;
                OUTPUT Range TABLE "RA0", "RA1", "RA2", "RA3", "RA4";
        END ACTIONS;
        GET ACTIONS;
                OUTPUT STRING "RA?";
                ENTER Range FORMAT 'K';
        END ACTIONS;
END COMPONENT;
```

(Continued)

```
PANEL Multimeter;
        POSITION 1,1;
        SIZE 214,213;

        DISPLAY Reading;
                POSITION 80, 180;
                TITLE "Reading";
                FORMAT "5DIGITS";
        END DISPLAY;

        DISCRETE Function;
                POSITION 80, 150;
                TITLE "Function";
        END DISCRETE;

        DISCRETE Range;
                POSITION 80, 120;
                TITLE "Range";
        END DISCRETE;
END PANEL;
```



Reading

Function DCV

Range 30mV

Figure 1-1.
A Simple Panel for a Multimeter

Here are some important points to remember about HP IDs:

- You must have a REVISION statement at the top.
- Component declarations come before panel declarations.
- Component declarations do not *nest* one inside the other, but panel declarations do. This is because a panel may contain other *subpanels* (panels within panels).
- The typical SET ACTIONS is just an output statement.
- The typical GET ACTIONS is an output or a query followed by an `ENTER`.
- If additional programmatic power is needed for a more complicated instrument, the actions can become quite advanced, including IF/THEN/ELSE and SELECT/CASE statements, calls to subprograms, and so on.

2

The Component Section

Overview

The component section is the portion of the HP ID that controls the instrument.

The component section is divided into two parts:

1. General statements that specify typical attributes of the HP ID. These include EOL (end of line), initialization, error detection, command prefix information, version information, and custom state control. You can have one of each of the general statements anywhere in the component section.
2. A series of compound statements. The statements within each compound statement describe the component and include one or more sets of action lists. You can have as many components statements as you want.

Action lists are compound statements in which you provide a list of statements that the HP ID executes such as an ENTER.

An action list may be defined in a component description as a SET or GET action or separately outside of a component as a named action list.

In general, action statements primarily affect instrument I/O and component interactions.

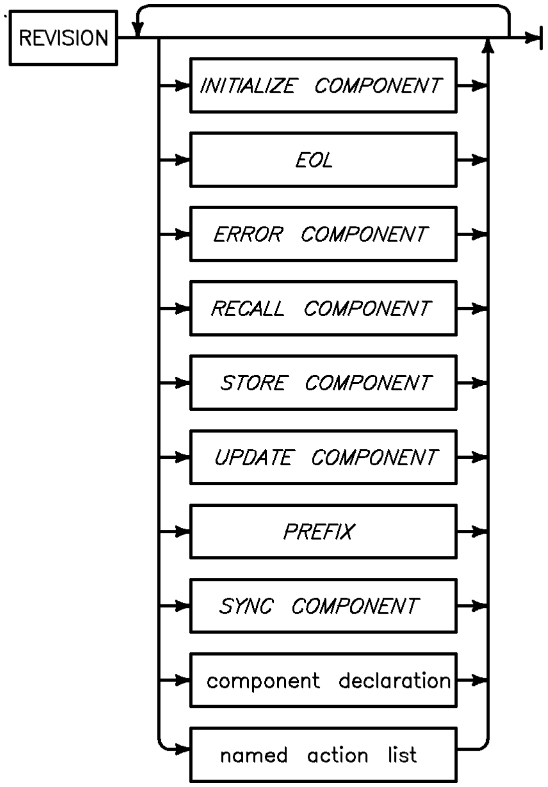


Figure 2-1. Syntax for the Component Section

General Statements

There are nine general statements, of which only the REVISION is required. HP suggests that all HP IDs use the INITIALIZE COMPONENT and ERROR COMPONENT statements. Use of the other general statements is optional, depending on the particular instrument involved. These general statements are:

- The REVISION statement.

  This is the required statement and must be the first non-commented statement of the HP ID. REVISION is used to keep track of the version of the HP ID Language.

- The INITIALIZE COMPONENT statement.

  Allows you to specify a component whose SET ACTIONS are executed whenever the user loads the HP ID or pushes the "Reset" button.

- The EOL statement.

  Allows you to specify an OUTPUT end-of-line (EOL) sequence if your instrument requires one that differs from the default. The default is carriage return, line feed, no EOI.

- The ERROR COMPONENT statement.

  Allows the HP ID to query the instrument to see if there was an error after each instrument transaction.

- The RECALL COMPONENT statement.

  Allows you to specify a component whose SET ACTIONS are executed at the beginning of a state recall.

- The STORE COMPONENT statement.

  Allows you to specify a component whose SET ACTIONS are executed when a state is stored.

- The UPDATE COMPONENT statement.

  Allows you to specify a component whose GET ACTIONS will be executed repetitively as long as there is no cursor movement or other actions lists being executed. The UPDATE component is generally used to provide interactive updates on reading or XY displays.

- The PREFIX statement.

  Allows you to address cards in a card-cage instrument. The PREFIX string is sent to the instrument before a SET action or GET action is executed.

- The SYNC statement.

  Allows you to specify a component whose GET ACTIONS will be executed whenever the user of the HP ID requests that the HP ID "sync up" with the current state of the instrument. The HP ID should attempt to query the state of the instrument, updating the HP ID without changing the instrument.

Components and Status

A component is a place holder for a value, much like a variable in BASIC or C. Component declarations contain more than just a variable declaration, however.

A component also has a status associated with it. A component's status is always one of the following three:

- Valid (VALID).
- Invalid (INVALID).
- Don't care (DONTCARE).

The value of a component and its status are independent. Generally, if a component is VALID, its value matches that of the corresponding control in the instrument. If a component is INVALID, its value may not be the same as that of the corresponding control. If a component is DONTCARE, the value of the corresponding control is currently not important to the operation of the instrument.

The component description must contain the following:

- A name.
- The type of data that will be associated with the component.

A component description may also contain the following:

- The valid values and an initial value.
- The actions necessary to control the instrument (that is, send and receive information).
- A list of components that are *coupled* to the current component (that is, their value depends on the value of the current component).

The following table illustrates some of the attributes of a component.

Table 2-1. Attributes of a Component

Instrument Feature	Component Name	Component Type	Valid Values	Initial Value
DMM function	FUNCTION	DISCRETE	DCV, ACV, ...	DCV
Source frequency	FREQUENCY	REAL	0 - 20E6	10000.0
DMM reading	READING	REAL	—	—
Oscilloscope waveform	TRACE	RARRAY	—	—

A Component Description

A component description is a compound statement (see figure 2-2) containing a series of statements that specify the attributes of the component. A component description begins with the keyword COMPONENT and ends with END COMPONENT.

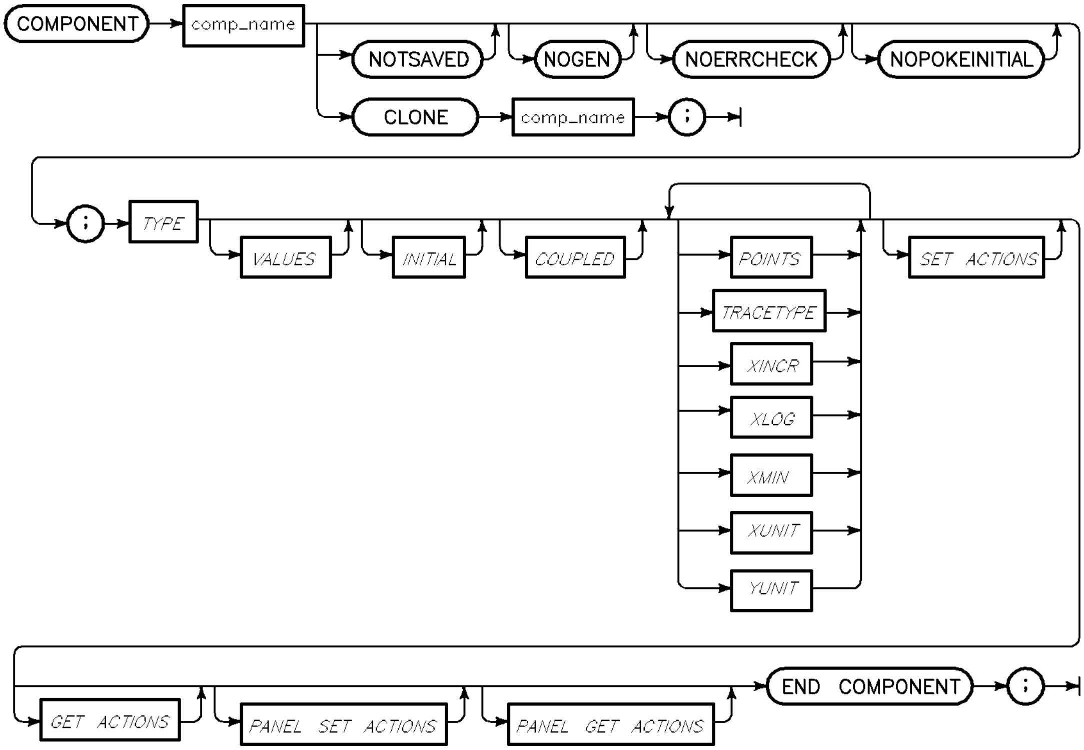


Figure 2-2. Syntax of a Component

Requirements

Each component description must have a name immediately following the keyword COMPONENT. All components except those specified as CLONE must have a TYPE statement. The rest of the statements are optional (except TYPE DISCRETE components, for which VALUES is required).

Name

The component name must start with an alpha character, A-Z or a-z. After that character, it may be followed by any mix of alphanumeric characters, either upper or lower case, or by an underscore. The syntax in not case sensitive.

Immediately after the name, you can include the following optional keywords:

- CLONE
    - □ CLONE specifies that the component is identical to another component in all attributes except name. Following the keyword CLONE is the name of the component that is being cloned. A CLONE component description is just a single statement. For example:

      ```
      COMPONENT Switch2 CLONE Switch1;
      ```

    - □ If the component `Switch1` is specified as NOTSAVED, NOGEN, NOPOKEINITIAL, and/or NOERRCHECK, then the component `Switch2` is defined the same way. Additionally, all actions are identical.
- NOTSAVED

  NOTSAVED specifies that the HP ID should not include the value of the component when the user stores or saves a panel setup as a state in the development environment. If a component is not specified as NOTSAVED, the HP ID always saves its value. NOTSAVED is typically used on

    - □ dummy components that contain a value but do not relate to any setting in the instrument
    - □ menu components that change panels but do not have any impact on the instrument
    - □ components that implement some automatic setup feature of the instrument
    - □ components that hold readings returned from the instrument
- NOGEN

  NOGEN is applicable to HP ITG only.

  NOGEN specifies that the HP ID should not generate any code that references the component. If a component is not specified as NOGEN, HP ITG generates code in the editor window involving the component when the user has turned on Log HP ITG Calls. Again, menu components and dummy components should be NOGEN.

- NOERRCHECK

  NOERRCHECK specifies that the HP ID should not perform default error checking after executing any actions initiated by this component. For example, components that set up triggering in multimeters may want to avoid an error check, since the bus traffic for the error check might cause unpredictable triggers (see “ERROR COMPONENT” in chapter 8, “Component/Action Syntax”).

- NOPOKEINITIAL

  NOPOKEINITIAL specifies that the POKEINITIAL statement (usually present in the INITIALIZE COMPONENT) does not affect this component. That is, normally a POKEINITIAL statement will set every component to its initial value. If a component specifies NOPOKEINITIAL, then execution of a POKEINITIAL will not affect the value of this component. NOPOKEINITIAL is useful only in rare situations on complex instruments where a RESET of the instrument does not reset the value of a particular component.

Type Statement

The HP ID uses the information in the TYPE statement to determine how much memory to allocate for the component’s value. You can select from the following types:

- DISCRETE

  The component can have a value that is selected from a predefined list of labels.

- INTEGER

  The component can have a value that is a 16-bit integer, from -32768 to 32767.

- CONTINUOUS

  The component can have a value that is a 64-bit real number.

- STRING

  The component can have a string value that is not more than 256 characters.

- IARRAY

  The component can have a value that is a one- or two-dimensional array of integers.

- RARRAY

  The component can have a value that is a one- or two-dimensional array of real numbers.

- ITRACE

  The component can have a value that is a one- or two-dimensional array of integers. It also may have further information on attributes of the data.

- RTRACE

  The component can have a value that is a one- or two-dimensional array of reals. It also may have further information on attributes of the data.

VALUES Statement

The VALUES statement allows you to specify a list or range of allowable values for the component. When the user is using the HP ID in the development environment, the VALUES statement restricts the values that the user can enter. The VALUES statement is required for TYPE DISCRETE components where it is used to list the allowed values.

RANGE

For INTEGER and CONTINUOUS type components, you can specify a RANGE that marks the lowest and highest allowable values.

Part of the RANGE specification includes a step size that can be either linear or logarithmic. This step size is used to round the value given by the user to the nearest legal value for the instrument.

An optional part of the RANGE specification is AUTO. You can include this keyword at the end of the RANGE specification if your instrument has an autoranging feature associated with this component.

INITIAL Statement

The INITIAL statement allows you to specify an initial value for the component as well as an initial status, for example, INVALID or DONTCARE.

When a POKEINITIAL statement is executed in an action list, the HP ID sets all the components with an INITIAL statement to their INITIAL value and status, unless NOPOKEINITIAL is specified.

If a component doesn't include an INITIAL statement, the default status is VALID and the default value is determined by these rules:

- DISCRETE components are set to the first value in the VALUES list.
- STRING components are set to " ", that is, null.
- All other components are set to 0.

COUPLED Statement

The COUPLED statement is designed to simplify handling the relationship between different components. COUPLED is used when changing the value of one component, like center frequency, might change the value other other components, like start and stop frequency. In the COUPLED statement, you specify the name of one or more components. When the HP ID executes the SET ACTIONS of the component containing the COUPLED statement, it does the following:

- In the development environment, executes the GET ACTIONS of all components specified in the COUPLED statement.
- In the run-time environment, invalidates components specified in the COUPLED statement.
- During a recall state, COUPLED has no effect.

In general, if you have two components, A and B for example, and changing the value of A may change the value of B, you should include the statement

```
COUPLED B;
```

in the A component's description.

SET ACTIONS Compound Statement

The HP ID executes the statements in this action list when the component's value is set (that is, when the user changes a component's value). If the component is associated with a control, it usually needs at least a SET ACTIONS list.

GET ACTIONS Compound Statement

The HP ID executes the statements in this actions list when the instrument is requested to provide a value for the component (that is, when the user asks the instrument for this value). If the component represents a measurement or query, it usually needs only a GET ACTIONS list.

PANEL SET ACTIONS Compound Statement

The HP ID executes the statements in this action list after it executes the component's SET ACTIONS list, if there is one. If there is not a SET ACTIONS list, then the HP ID executes the PANEL SET ACTIONS list in its place. The HP ID executes this list only in the development environment.

The PANEL SET ACTIONS list is provided to perform extra actions that make the panel easier to use. These actions do not need to be executed when you run programs you develop with the development environment.

For example, if the instrument might round the value, then a PANEL SET ACTIONS might look like

```
PANEL SET ACTIONS;
  GET Compname;
END ACTIONS;
```

which would query the rounded value from the instrument so that the panel is correct. During run time, this is not needed, so putting it in a PANEL SET ACTIONS list avoids the overhead of the query.

PANEL GET ACTIONS Compound Statement

The HP ID executes the statements in this action list after it executes the component's GET ACTIONS list, if there is one. If there is not a GET ACTIONS list, then the HP ID executes the PANEL GET ACTIONS list in its place. The PANEL GET ACTIONS are only executed in the development environment.

The PANEL GET ACTIONS list is provided to perform extra actions that make the panel easier to use but do not need to be executed when you run programs generated from the development environment.

What a State is

A state is a collection of the values and status of each component in the HP ID that is needed to set up the instrument. In our multimeter example in chapter 1, a state might be as follows:

Table 2-2.

Component	Value	Status
Reading	?	INVALID
Range	3V	VALID
Function	ACV	VALID

This is one state of the instrument, and we can store this state and give it a name, for example, "AC_3V". In this state, the Range is 3V, and the status is VALID, so we expect that the Range component and the Range control in the instrument are the same. Reading, however, is INVALID, and so we say that we don't know what the value is, and the display shows "?".

Notice that Reading is not important for setting up the state of the instrument. We do not send Reading to the instrument in order to set it up for a different measurement. Therefore, we can put the word NOTSAVED in the component declaration for Reading, which means that Reading is not part of the state of the instrument.

```
COMPONENT Reading NOTSAVED;
  ...
END COMPONENT;
```

What Happens During Store State

When a state is stored, a list is built with the value and status of each component that is not NOTSAVED. So, let's say that we store two states, one ACV, 30V and one DCV, 3V. HP ID writers need not concern themselves with maintaining this list.

What Happens During Recall State

Assume that the multimeter is in state ACV, 30V and we want to go to state DCV, 3V. So, we recall the state DCV, 3V.

The saved components are scanned in the order they appear in the HP ID, marking as INVALID those whose values differ from the desired value in the desired state. Both Function and Range are marked as INVALID here. Reading is specified as NOTSAVED, therefore, it is skipped. Next, recall makes another pass through each saved component and executes a SET on each component marked as INVALID during the first pass. This executes the SET ACTIONS statements in that component, causing the HP ID to send `FN0` and `RA2` to the multimeter; this also causes Function and Range to be marked as VALID. The multimeter is now in the state DCV, 3V.

Suppose that we recall a state of DCV, 30V. Then recall goes through the same two passes of the components, but this time Function is already set to the desired value, so it is not marked as INVALID. On the second pass, only the new range is sent, `RA3`.

This example demonstrates the power of state recall in reducing bus traffic by tracking the state of the instrument and sending the fewest HP-IB commands needed to achieve a state transition. This allows the test developer to define the states needed in the application without regard to the previous or next state that will be needed. The HP ID sends the fewest commands necessary to reach the new state regardless of the current state.

Note For complete information on recalling a state, refer to "How Recall Works" in chapter 6, "Advanced Topics."

3 The Panel Section

Overview

The panel section is the portion of the HP ID that builds a panel in the development environment. When a program developer uses the HP ID in the development environment, the interactions with the panel control the instrument and generate instrument-control code.

When in a run-time mode, the HP ID does not execute the action lists associated with the panels (such as UPDATE ACTIONS and HIT ACTIONS). This means that there is no overhead associated with the panel section of the HP ID and the program executes faster. It follows, then, that UPDATE and HIT ACTIONS should not be used for anything necessary in the run-time environment.

Structure

The panel section follows the component section in the HP ID. The panel section is one large compound statement that begins with PANEL and ends with END PANEL.

Within the PANEL ... END PANEL compound statement, you add statements to design the panel (see chapter 9, "Panel Syntax"). These statements are divided into three categories:

- **Attributes**
  - □ Attributes includes the name, size, and position of the panel, as well as text and fill colors.
  - □ The HP ID Language provides defaults for most of these attributes so that you need specify them only if you wish to vary from the default.

- **Panel elements**
  - □ These are predefined elements that you use to build a panel. They include text fields, various controls, and displays.
  - □ The panel elements are compound statements in which you can specify the attributes of the element.
  - □ These attributes vary from element to element. The HP ID Language provides defaults for most of these attributes so that you specify them only if you wish to vary from the default.
  - □ The name you give to a panel element links that element to a component whose action lists are executed when the user interacts with the element.
- **Actions**
  - □ There are two action lists that can be used with certain panel elements.
  - □ If a HIT ACTIONS list is specified, the HP ID executes this list instead of its default behavior for the panel element.
  - □ If an UPDATE ACTIONS list is specified, the HP ID executes this list whenever the view of the associated component changes.

You can nest PANEL ... END PANEL statements within the main one, creating a multi-layered panel. You can use all the panel elements in both the main and nested PANEL ... END PANEL statements. The nested panels are called **subpanels** in this manual.

The Panel Elements

The HP ID Language provides a set of predefined elements from which you select when designing a panel. These elements fall into three categories:

- **Text fields:** These allow you to generate text on a panel. You can use this capability to label the other elements, increasing the overall friendliness of the panel.
- **Controls:** These allow the user to control an instrument and generate instrument-control code.

- **Displays:** These provide a way for you to display data from an instrument.

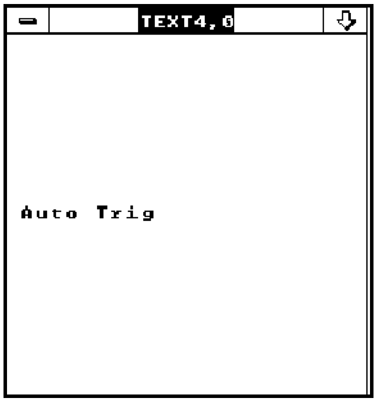
Usually, when you use a panel element other than TEXT, you'll need to associate it with a component description. This means that the name following the panel element must be the same as a component name of the right TYPE. Many panel elements can only be associated with certain TYPEs of components. A panel element provides a view of the value of the component.

You can use the same component for several different panel elements. For example, you can create a Frequency control that appears on five different subpanels. In the HP ID, there would be one component description for Frequency, but it would be referenced in a panel element description in five subpanels.

Text

TEXT is the only panel element in this category. When you use this element, specify the text and where you want it displayed on the panel. You can also specify the font, area-fill color, and line color if you don't want to use defaults. TEXT is usually used to label the controls and displays on the panel. TEXT does not require a matching component name.

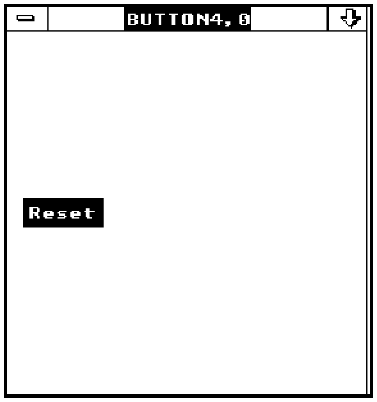
```
REVISION 2.0;

PANEL MAIN;
  TEXT "Auto Trig";
     POSITION 8,98;
  END TEXT;
END PANEL;
```



Controls

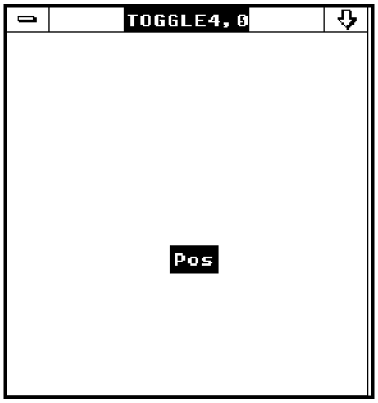
The **BUTTON** control allows you to create a push button on an instrument panel. When a BUTTON is pushed, the HP ID immediately executes the SET ACTIONS associated with that button. BUTTON requires a matching INTEGER component. You can include a HIT ACTIONS list in a BUTTON ... END BUTTON compound statement.

```
REVISION 2.0;

COMPONENT Reset;
  TYPE INTEGER;
END COMPONENT;

PANEL MAIN;
  BUTTON Reset;
    POSITION 10,98;
    LABEL "Reset";
  END BUTTON;
END PANEL;
```



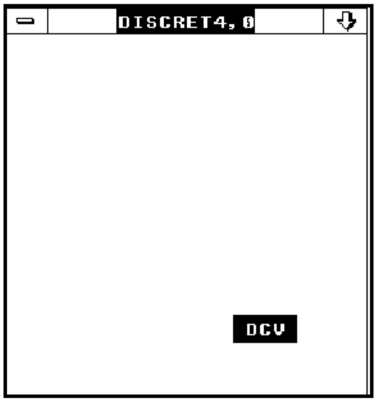
The **TOGGLE** control is a two-position control. It selects one of the two values available for instrument controls that have two values. *Pushing* the TOGGLE button toward a setting executes that setting's SET ACTIONS. TOGGLE requires a matching DISCRETE component.

```
REVISION 2.0;

COMPONENT SlopeA;
  TYPE DISCRETE;
  VALUES POS,NEG;
END COMPONENT;

PANEL MAIN;
  TOGGLE SlopeA;
    POSITION 96,70;
    LABEL "Pos","Neg";
  END TOGGLE;
END PANEL;
```



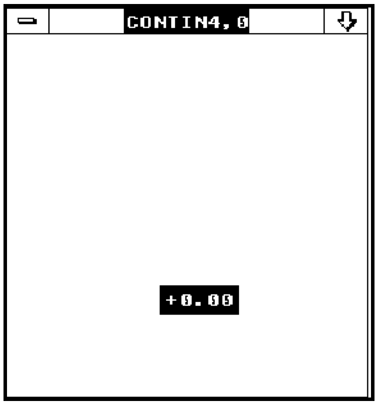
The **DISCRETE** control allows you to specify a set of valid values from which the user can select. DISCRETE requires a DISCRETE component.

```
REVISION 2.0;

COMPONENT Function;
  TYPE DISCRETE;
  VALUES DCV,ACV,OHMS;
END COMPONENT;

PANEL MAIN;
  DISCRETE Function;
    POSITION 134,29;
    LABEL "DCV","ACV","OHMS";
  END DISCRETE;
END PANEL;
```



The **CONTINUOUS** control allows you to specify a range of valid values. CONTINUOUS requires either an INTEGER or CONTINUOUS component.

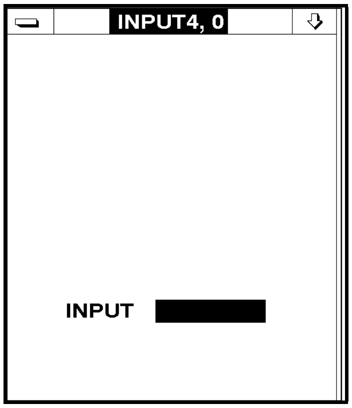
```
REVISION 2.0;

COMPONENT IntTrigLevelA;
  TYPE CONTINUOUS;
END COMPONENT;

PANEL MAIN;
  CONTINUOUS IntTrigLevelA;
    POSITION 90,47;
    STYLE "NOENGR";
    FORMAT "SD.DD";
  END CONTINUOUS;
END PANEL;
```



+0.00

The **INPUT** control allows you to create a field in which the user can enter text. INPUT requires a STRING, CONTINUOUS, or INTEGER component.

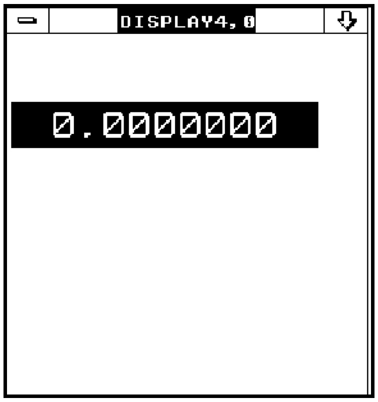
```
REVISION 2.0;

COMPONENT UserInput;
  TYPE STRING 10;
END COMPONENT;

PANEL MAIN;
  INPUT UserInput;
    POSITION 90,47;
    TITLE "Input";
  END INPUT;
END PANEL;
```



Displays

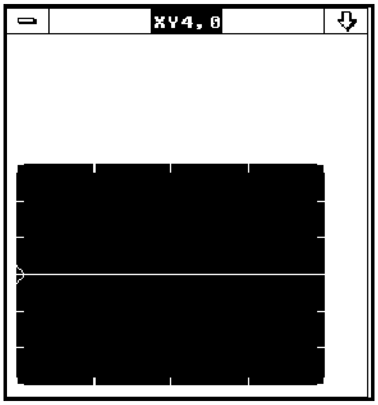
The **DISPLAY** panel element provides one or more lines where the display can show measurement or status data from the instrument. DISPLAY requires a CONTINUOUS, INTEGER, DISCRETE, or STRING component.

```
REVISION 2.0;

COMPONENT Reading;
  TYPE CONTINUOUS;
END COMPONENT;

PANEL MAIN;
  DISPLAY Reading;
    POSITION 2,145;
    FONT 15,25;
    FORMAT "D.DDDDDDD";
  END DISPLAY;
END PANEL;
```



The **XY** element displays array data from the instrument. You can optionally specify a matching INTEGER component. You can include a HIT ACTIONS list in an XY ... END XY compound statement.

```
REVISION 2.0;

COMPONENT TraceA;
  TYPE RARRAY 1,5;
END COMPONENT;

COMPONENT Acquire;
  TYPE INTEGER;
END COMPONENT;

COMPONENT Current;
  TYPE INTEGER;
END COMPONENT;

PANEL MAIN;
  XY Acquire;
    POSITION 5,5;
    SCALE 1,5,-3,3;
    GRATICULE FRAME;
    TRACE TraceA;
      MARKER Current;
        TYPE POINT;
      END MARKER;
    END TRACE;
  END XY;
END PANEL;
```



Designing a Panel

This section discusses several of the most important attribute statements. Please refer to chapter 9, “Panel Syntax,” for complete information on all the panel statements.

The following discussion is fairly general. Note that there are subtle differences in the attribute statements, depending on the panel element.

Panel Layout

The HP ID generates a title bar that spans the top of each instrument panel. This title bar contains the name the user enters when adding the panel to a soft test system as well as the HP-IB address of the instrument. The title bar also contains two icons:

- A box on the left. Clicking on it displays the panel menu.
- An arrow on the right. Clicking on it reduces the panel to an icon.

Size of Panel Elements

The SIZE statement allows you to specify the size of a panel or panel element.

The default size of a panel is 214 by 213 pixels.

The HP ID determines the default size of a panel element based on the font and length of the longest item that will be displayed. For example, to determine the default size of a BUTTON, we use the following formulas:

width = (LEN(LABEL) × font_width) + 4
height = font_height + 4

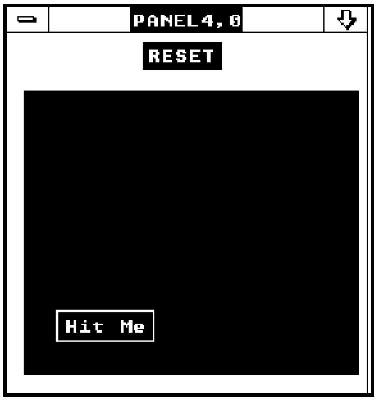
LABEL is the text that is displayed inside the BUTTON. The default font width is 9; the default font height is 15. If you do not want to use the font defaults, you can specify a width and height using the FONT statement.

Some panel elements accept the FORMAT and STYLE statements. These allow you to specify how you want numbers displayed. FORMAT requires a string in which you insert image specifiers. The STYLE statement allows you to specify that engineering prefixes not be used (for example, k for kilo, m for milli). When used, both the FORMAT and STYLE statements may affect the default size of the panel element.

Positioning Panel Elements

The POSITION statement allows you to specify the position of either a subpanel on the main panel or a panel element on the main panel or subpanel. Specify the position in pixels relative to the lower left corner of the panel.

The default position for most panel elements is 1,1, meaning one pixel over and one pixel up from the bottom left corner of the panel. Obviously, if you use the default position for more than one panel element, the elements will overlap on the panel. Therefore, you should plan to specify a position for each panel element.



Using Color

You can use the the FOREGROUND and BACKGROUND statements to specify the area-fill, text, and border colors for the panel or panel element. However, the default colors are recommended for consistency between panels.

General Guidelines

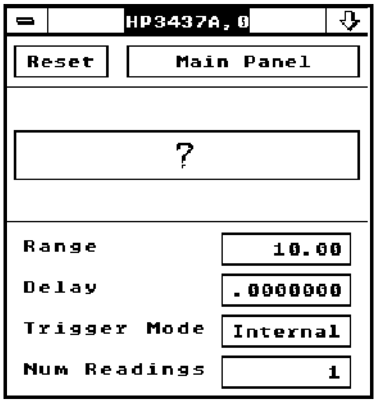
Hewlett-Packard recommends the following guidelines, which were established during the development of the HP IDs:

- Position displays in the upper left portion of the panel.
- For XY displays, position any controls associated with the XY display to the right of the display or across the bottom if there is room. (See "XY" in chapter 9, "Panel Syntax" for information on the default colors for the XY traces.)
- All controls except BUTTONs and TOGGLEs should be identified by a TEXT or TITLE field to their left.
- Control boxes should be at least three pixels apart so they don't appear too close together.

Other HP Conventions

All HP IDs produce panels that provide a Reset BUTTON, which is located in the top left portion of the panel, above any displays. The Reset component that is associated with the Reset BUTTON contains a SET ACTIONS list that contains a POKEINITIAL statement. When the user clicks on this button, the HP ID configures the instrument and the HP ID to the specified initialization state (see chapter 6, "Advanced Topics").

Most HP IDs also provide a menu control in the top right portion of the panel. You can use this menu control to go from subpanel to subpanel, with each subpanel providing a different set of controls and displays. See the next section, "Creating a Multi-Layered Panel," for information on creating a multi-layered panel.



```
Range              10.00
Delay           .0000000
Trigger Mode    Internal
Num Readings           1
```

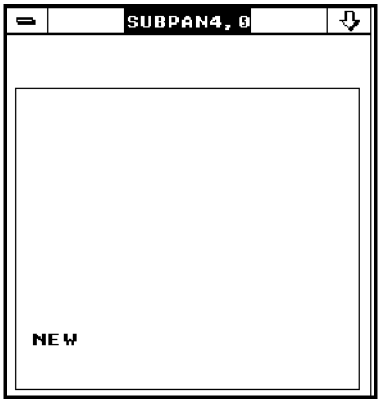
Creating a Multi-Layered Panel

An instrument panel can include one or more subpanels, allowing you to create a multi-layered panel. This is valuable because most instruments contain many more functions than can fit on one panel. Subpanels allow you to replace one set of controls and displays with another.

As mentioned earlier, the panel section really consists of one large PANEL compound statement. In the following example, `Parent` is the name of the main panel, and `Sub1` is a subpanel of `Parent`.

The subpanel containing text, NEW, is the default size for subpanels.

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;
COMPONENT Reset;
  TYPE INTEGER;
  SET ACTIONS;
    SHOW Sub1;
  END ACTIONS;
END COMPONENT;

PANEL Parent;
  PANEL Sub1;
    FOREGROUND (0,0,0) 0;
    TEXT 10,20,"NEW";
  END PANEL;
END PANEL;
```



Displaying and Hiding Subpanels

The SHOW and HIDE action statements control which subpanels are displayed in the development environment.

When you specify a panel in a SHOW statement, the subpanel is displayed if the parent panel is currently being shown. When the HP ID executes a HIDE statement, it hides the subpanel specified. The main panel can never be hidden because there is no outer panel in which it is nested.

Note To display a subpanel when a panel is first loaded into the development environment, use the INITIALIZE COMPONENT statement.

Moving Between Subpanels

You can use either a BUTTON or DISCRETE control to allow the user to access the different subpanels. When you use a BUTTON control, the subpanel is displayed when the user clicks on the button. When you use a DISCRETE control, a listbox is displayed from which the user can select a subpanel to be displayed.

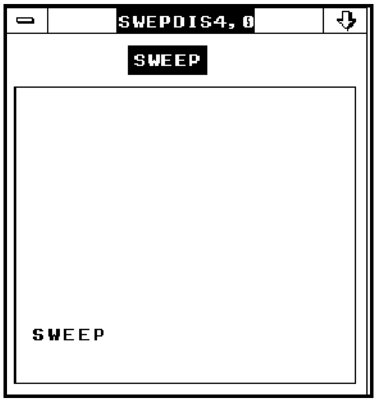
Using a DISCRETE Control

Here is the code needed to implement a Sweep subpanel and a Phase subpanel using a DISCRETE control.

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;
COMPONENT Reset;
  TYPE INTEGER;
  SET ACTIONS;
    POKEINITIAL;
    SET Menu;
  END ACTIONS;
END COMPONENT;

COMPONENT Menu;
  TYPE DISCRETE;
  VALUES Sweep,Phase;
  INITIAL Sweep;
  SET ACTIONS;
    SELECT Menu:
      CASE Sweep;
        HIDE Ph_panel;
        SHOW Sw_panel;
      CASE Phase;
        HIDE Sw_panel;
        SHOW Ph_panel;
```

(Continued)

```
    END SELECT;
  END ACTIONS;
END COMPONENT;

PANEL Main;
  PANEL Sw_panel;
    TEXT 10,20,"SWEEP";
    FOREGROUND (0,0,0) 100;
  END PANEL;
  PANEL Ph_panel;
    TEXT 10,20,"PHASE";
    FOREGROUND (0,0,0) 100;
  END PANEL;
  DISCRETE Menu;
    POSITION 72,188;
  END DISCRETE;
END PANEL;
```



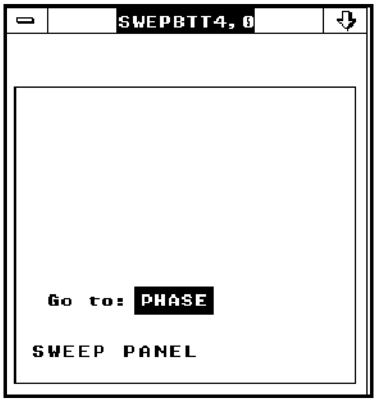
Using a BUTTON Control

Here is the code needed to implement a Sweep subpanel and a Phase subpanel using a BUTTON control.

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;
COMPONENT Reset;
  TYPE INTEGER;
  SET ACTIONS;
    SHOW Sw_panel;
  END ACTIONS;
END COMPONENT;

COMPONENT Phase;
  TYPE INTEGER;
  SET ACTIONS;
      HIDE Sw_panel;
      SHOW Ph_panel;
  END ACTIONS;
END COMPONENT;

COMPONENT Sweep;
  TYPE INTEGER;
  SET ACTIONS;
    HIDE Ph_panel;
    SHOW Sw_panel;
  END ACTIONS;
END COMPONENT;

PANEL Main;
  PANEL Sw_panel;
    TEXT 10,10,"SWEEP PANEL";
    FOREGROUND (0,0,0) 100;
```

(Continued)

```
    BUTTON Phase;
      POSITION 70,40;
      TITLE "Go to:";
    END BUTTON;
  END PANEL;

  PANEL Ph_panel;
    TEXT 10,20,"PHASE PANEL";
    FOREGROUND (0,0,0) 100;
    BUTTON Sweep;
      POSITION 70,40;
      TITLE "Go to:";
    END BUTTON;
  END PANEL;
END PANEL;
```



4

Action Lists

Overview

An **action list** is a compound statement in which you specify the actions that the HP ID must execute. These action lists are executed when SETs or GETs are done on a component.

You can call other action lists from within an action list, as well as specify general action lists that can be called from any action list. This is necessary because most instruments have interrelated functions.

There are seven places where you can use an action list:

1. Named outside of a component.
2. Unnamed SET ACTIONS in a component.
3. Unnamed GET ACTIONS in a component.
4. Unnamed PANEL SET ACTIONS in a component.
5. Unnamed PANEL GET ACTIONS in a component.
6. Unnamed HIT ACTIONS in a panel element.
7. Unnamed UPDATE ACTIONS in a panel element.

Creating action lists accounts for the majority of the time spent creating an HP ID. The HP ID Language provides over 20 action statements that you can use in an action list. These statements cover the following:

- instrument I/O
- component interactions
- wait times and conditions
- HP-IB controls
- subprogram and subroutine calls
- stack operations and calculations
- panel control
- program flow
- miscellaneous

Named ACTIONS

The ACTIONS ... END ACTIONS compound statement allows you to define an action list that can be associated with many different components or panel elements.

You can substitute the ACTIONS name for an actions list when you create any unnamed action lists. For example, if two components require the same SET ACTIONS list you could use the following code rather than writing them out twice.

```
ACTIONS Common_actions;
        OUTPUT A FORMAT K;
        OUTPUT B FORMAT K;
END ACTIONS;

COMPONENT A;
        TYPE INTEGER;
        SET ACTIONS Common_actions;
END COMPONENT;
COMPONENT B;
        TYPE INTEGER;
        SET ACTIONS Common_actions;
END COMPONENT;
```

Within an actions list, you can use the GOSUB action statement to call an action list. For example:

```
ACTIONS Freq0;
        FETCH 0;
        STORE FREQ;
END ACTIONS;

COMPONENT A;
        TYPE INTEGER;
        SET ACTIONS;
             OUTPUT A FORMAT K;
             GOSUB Freq0;
        END ACTIONS;
END COMPONENT;
```

SET ACTIONS Statement

The SET ACTIONS ... END ACTIONS compound statement can be used in a component description; it allows you to create a SET ACTIONS list. The action statements in this list are executed when a SET is done from the HP ID Language, or when the user of the HP ID changes the value of the component, or when state recall must change the value of the component.

Here's an example of a SET ACTIONS list:

```
COMPONENT Function;
        TYPE DISCRETE;
        VALUES DCV,ACV;
        INITIAL DCV;
        SET ACTIONS;
                OUTPUT Function TABLE "FN0","FN1";
        END ACTIONS;
        GET ACTIONS;
                OUTPUT STRING "FN?";
                ENTER Function FORMAT 'K';
         END ACTIONS;
END COMPONENT;
```

In the following example, a named actions list is specified in the SET ACTIONS statement.

```
ACTIONS Common_actions;
  OUTPUT IMPEDANCE_1 FORMAT K;
END ACTIONS;
!
COMPONENT IMPEDANCE_1;
  TYPE DISCRETE;
  VALUES ON,OFF;
  SET ACTIONS Common_actions;
END COMPONENT;
```

GET ACTIONS Statement

The GET ACTIONS ... END ACTIONS compound statement can be used in a component description; it also allows you to create a GET ACTIONS list. The action statements in this list are executed when the user of the HP ID asks for the value of a component, or when a GET is done from the HP ID Language itself, or when a COUPLED statement causes a GET.

You are responsible for including the action statements necessary to query the instrument or instruct it to make a measurement and return the value. You are also responsible for calling any other action lists that must be executed as a result of the value returned.

Here's an example of a GET ACTIONS list:

```
COMPONENT Function;
        TYPE DISCRETE;
        VALUES DCV,ACV;
        INITIAL DCV;
        SET ACTIONS;
                OUTPUT Function TABLE "FN0","FN1";
        END ACTIONS;
        GET ACTIONS;
                OUTPUT STRING "FN?";
                ENTER Function FORMAT 'K';
        END ACTIONS;
END COMPONENT;
```

In the following example, a named actions list is specified in the GET ACTIONS statement:

```
ACTIONS Common_actions;
        OUTPUT STRING "FN?";
        ENTER Function FORMAT 'K';
END ACTIONS;
COMPONENT Function;
        TYPE DISCRETE;
        VALUES DCV,ACV;
        INITIAL DCV;
        SET ACTIONS;
                OUTPUT Function TABLE "FN0", "FN1";
        END ACTIONS;
        GET ACTIONS Common_actions;
END COMPONENT;
```

PANEL SET ACTIONS

The PANEL SET ACTIONS ... END ACTIONS compound statement can be used in a component description, allowing you to create a PANEL SET ACTIONS list. PANEL SET ACTIONS are executed after SET ACTIONS.

Use this action list to perform actions needed to keep the panel up-to-date. Since this action list is executed only in the development environment, these actions won't slow execution during run time. The most commonly used statements in a PANEL SET ACTIONS list include:

- SHOW and HIDE

  Allow you to specify which panels are displayed based on selections the user has made.

- NOTIFY

  Allows you to specify a message that is displayed when the user selects a particular control.

Here's an example of a PANEL SET ACTIONS list.

```
COMPONENT Function;
        TYPE DISCRETE;
        VALUES DCV,ACV;
        INITIAL DCV;
        SET ACTIONS;
            OUTPUT Function TABLE "FN0","FN1";
        END ACTIONS;
        PANEL SET ACTIONS;
            SELECT Functions;
                 CASE DCV;
                      HIDE Filter_panel;
                 CASE ACV;
                      SHOW Filter_panel;
            END SELECT;
        END ACTIONS;
END COMPONENT;
```

PANEL GET ACTIONS

The PANEL GET ACTIONS ... END ACTIONS compound statement can be used in a component description, allowing you to create a PANEL GET ACTIONS list. PANEL GET ACTIONS are executed after GET ACTIONS.

Use this action list to perform actions needed to keep the panel up-to-date. Since this action list is executed only in the development environment, these actions won't slow down execution during run time.

Here's an example of a PANEL GET ACTIONS list:

```
COMPONENT AUTO_RANGE;
  TYPE DISCRETE;
  VALUES ON,OFF;
!       .
!       .
!       .
  PANEL GET ACTIONS;
    GET RANGE;
  END ACTIONS;
END COMPONENT;
```

HIT ACTIONS

The HIT ACTIONS ... END ACTIONS compound statement can be used in the description of a BUTTON, DISPLAY, or XY panel element. The statement also allows you to create a HIT ACTIONS list.

Normally, when a user clicks on a BUTTON or DISPLAY, the HP ID performs the following:

- Generates an HP ITG subprogram call in the editor window if Log HP ITG Calls is on.
- Performs a SET on the component associated with the BUTTON and a GET on the component associated with the DISPLAY.

CODEGEN is available with the HP ITG soft test system only.

However, if the panel element includes a HIT ACTIONS list, the HP ID does not perform any of the actions listed above. With a HIT ACTIONS list specified, you must include the CODEGEN statement in the list if you want HP ITG to generate code when the user clicks on the panel element with Log HP ITG Calls on. If you want to set the value of the component, you must include the SET statement in the list.

A HIT ACTIONS list gives you complete control over what the HP ID does when the user uses the instrument panel. For example, in the listing below, the HIT ACTIONS list specifies the panel be hidden before the new value of the component is set.

Here's an example of a HIT ACTIONS list:

```
BUTTON OK_SWEEP_PANEL;

        POSITION 116,6;
        SIZE 58,19;
        LABEL "Done";
        HIT ACTIONS;
            HIDE SWEEP_PNL;
            SET OK_SWEEP_PNL;
            CODEGEN POKE TIMER;
            CODEGEN POKE SAMPLE_EVT;
            CODEGEN SET NRGDS;
        END ACTIONS;
END BUTTON;
```

Since HIT ACTIONS are not executed in the run-time environment, HP suggests that they be used only for HIDE, SHOW, and other operations not affecting the state of the instrument.

UPDATE ACTIONS

The UPDATE ACTIONS ... END ACTIONS compound statement can be used in the description of any panel element except XY and TEXT; it allows you to create an UPDATE ACTIONS list.

The HP ID only executes the UPDATE ACTIONS list for the panel element when the view of the component changes.

Here's an example of an UPDATE ACTIONS list.

```
DISCRETE FUNCTION;
  POSITION 80,150;
  UPDATE ACTIONS;
    SELECT FUNCTION;
      CASE DCV;
        HIDE Filter_panel;
      CASE ACV;
        SHOW Filter_panel;
    END SELECT;
  END ACTIONS;
END DISCRETE;
```

Since UPDATE ACTIONS are not executed in the run-time environment, HP suggests that they be used only for HIDE, SHOW, and other operations not affecting the state of the instrument.

Action Statements

Following is a description of common action statements and their implications.

Instrument I/O

OUTPUT and ENTER

The OUTPUT statement is used to send commands to the instrument.

The ENTER statement is used to input data from the instrument and assign the values entered to the components. ENTER is most commonly used in GET ACTIONS lists.

FLUSH

Normally, all output strings are buffered. The FLUSH statement causes everything in the buffer to be sent to the instrument. The following example causes the HP ID to send `FNO` as a separate bus transaction rather than sending `FNORAO` all at once.

```
ACTIONS Common;
        OUTPUT STRING "FNO";
        FLUSH;
        OUTPUT STRING "RAO";
END ACTIONS;
```

Component Interactions

Following is a description of common component interactions and their implications.

SET and GET

The SET and GET statements are used to execute the action lists of a component.

SET is typically used after a value is stored into a component. It executes the SET ACTIONS list of the specified component, then the PANEL SET ACTIONS, if present.

GET is typically used to request that the instrument send the value of a component. GET executes the GET ACTIONS list of the specified component, then the PANEL GET ACTIONS, if present.

POKEINITIAL

The POKEINITIAL statement sets the value and status of all components with initial values as specified in the INITIAL statement (see "INITIAL" in chapter 8, "Component/Action Syntax"). Usually, only the INITIALIZE COMPONENT uses POKEINITIAL.

INVALIDATE, VALIDATE, and DONTCARE

INVALIDATE, VALIDATE, and DONTCARE statements explicitly control the status of a component's value.

A component is always in one of the following states:

- **Valid:** The HP ID's value for the component is the same as the instrument's. A component usually becomes valid when its value is sent to the instrument.

  The VALIDATE statement allows you to specify that a particular component's value is valid.

- **Invalid:** The HP ID's value for the component is not necessarily the same as the instrument's value.

  The INVALIDATE statement allows you to specify that a particular component's value is invalid.

- **Don't care:** The value of the component is not part of the current instrument state.

  The DONTCARE statement allows you to specify that a particular component's value should not be used when recalling a state. You can reverse this by using a VALIDATE or INVALIDATE statement.

Wait Times and Conditions

Following is a description of common component wait times and conditions and their implications.

WAIT SPOLL BIT and WAIT TIME

WAIT SPOLL BIT and WAIT TIME statements allow you to create a HP ID that will provide for settling delays in the instrument if the instrument does not provide for them itself.

The WAIT SPOLL BIT statement causes the HP ID to delay execution of the next statement until it receives a serial poll bit true (ready bit) from the instrument.

The WAIT TIME statement causes the HP ID to wait a specified number of seconds before executing the next statement.

HP-IB Controls

Following is a description of HP-IB controls and their implications.

TRIGGER

The TRIGGER statement sends a group execute trigger (GET) to the instrument.

SPOLL

The SPOLL statement performs a serial poll of the instrument and pushes the result onto the stack.

CLEAR

The CLEAR statement sends a selected device clear to the instrument.

Subprogram Calls

Following is a description of subprogram calls and their implications.

GOSUB

The GOSUB statement allows you to call a named action list as a subroutine of an action list.

Stack Operations

Following is a description of stack operations and their implications.

Math, Logical, and String Operators

These commands form the basis of a stack machine. The operators obtain their operands from the stack and return the result to the stack. You can use the FETCH statement to specify a source whose value you want to put on top of the stack. You can use the STORE statement to get a value from the top of the stack and put it in a destination. Figure 4-1 shows all operators valid in the HP ID Language. Refer to "ACTIONS" in chapter 8, "Component/Action Syntax," for a description of these operators.

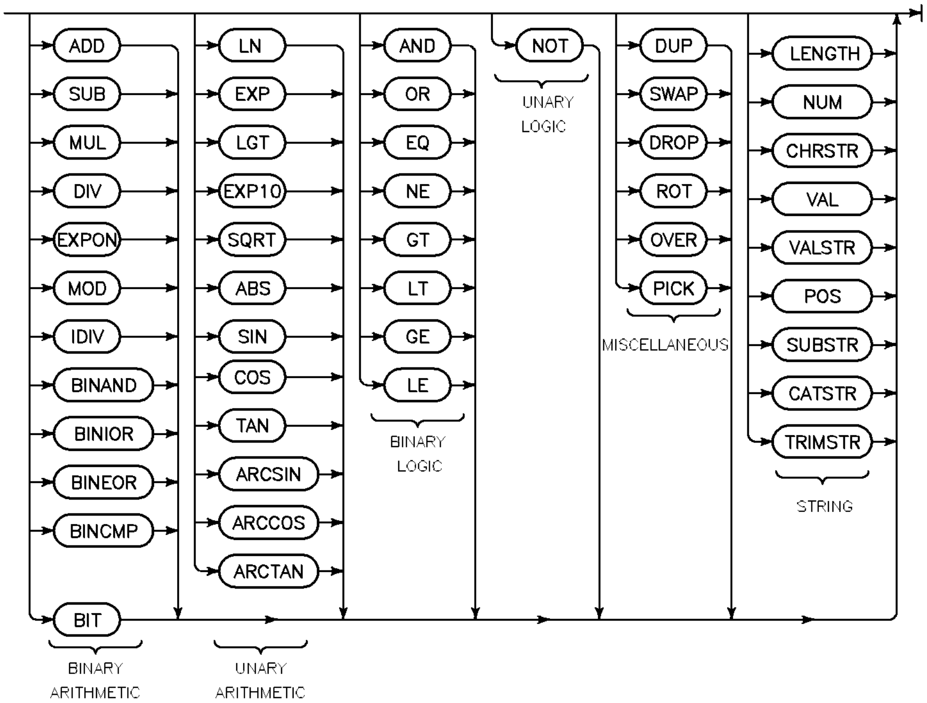


Figure 4-1. The HP ID Language Arithmetic, Logical, and String Operators

Panel Control

Following is a description of panel controls and their implications. Typically, these statements are used only in PANEL SET ACTIONS, PANEL GET ACTIONS, HIT ACTIONS, and UPDATE ACTIONS lists.

SHOW and HIDE

The SHOW and HIDE statements allow you to control which subpanels are displayed at any given time.

NOTIFY

The NOTIFY statement allows you to display a message to the user.

ENABLE and DISABLE

The ENABLE and DISABLE statements provide control over which values in a DISCRETE component are currently valid. For example, if your power supply does not have a relay option installed, you can disable the relay panel choice on the DISCRETE component that controls which panel is shown.

Program Flow

Following is a description of program flow control statements and their implications.

IF ... END IF, SELECT ... END SELECT, and LOOP ... END LOOP

The IF ... END IF statement provides conditional execution of alternate sections of code based on the specified value. If the value is not zero, then the THEN clause is executed; otherwise, the ELSE clause, if present, is executed.

The SELECT ... END SELECT statement allows for a multi-branch in code execution. Please refer to the syntax for the SELECT statement for complete information on which type of CASE can be used with which TYPE of components.

The LOOP and END LOOP statements define a loop structure that can be exited with one or more EXIT IF statements.

You can nest IF, SELECT, and LOOP statements ten deep.

Miscellaneous

Following is a list of miscellaneous information.

CODEGEN

CODEGEN is available with the HP ITG soft test system only.

The CODEGEN statement allows you to override the default code generation and specify which HP ITG subprograms HP ITG generates when the user selects a panel element and Log HP ITG Calls mode is on.

MATSCALE

The MATSCALE statement allows you to instruct the HP ID to perform mx+b arithmetic on the data in an array or track component, for example, RARRAY and ITRACE.

BITS

The BITS statement provides a way of building a bit pattern from a series of sources. The HP ID puts the extracted bit pattern on top of the stack. To build a bit pattern from separate sources requires an uninterrupted sequence of BITS statements.

5

Writing a Driver

Timeline in Writing a Driver

It takes time to write a useful HP ID. The time in writing, testing and documenting a good HP ID is directly proportional to the complexity of the instrument. You can expect to spend the following amount of time developing drivers for instruments with the following complexity:

- Simple instrument: 2-4 weeks (HP3437, HP5384 HP IDs)
- Moderately complex instrument: about 2 months (HP3314, HP3458 HP IDs)
- Complex instrument: about 3 months (HP54501, HP71000 HP IDs)

The design phase is the most important phase in the development of an HP ID. If the design phase is done well, it is fairly straightforward to code the HP ID.

Note ☝ The following definitions are used heavily in the text. An understanding of these terms and the distinctions between them will aid in reading the remainder of this manual.

instrument control - instrument controls reside on the instrument and affects instrument operation. A control is changed by using an interface, usually either the front panel or the HP-IB interface bus. A control usually has a set of valid settings, such as on or off, associated with it.

instrument command - an instrument command is a keyword or token sent over the HP-IB interface bus that causes the instrument to change a particular control to a new setting. Usually, but not always, an instrument command will be associated directly with a control on the instrument on a one-to-one basis.

instrument state - an instrument state is a set of values, each one corresponding to a setting of a particular control. Every control on the instrument is represented once and only once. The collection of all possible states of the instrument includes every operating configuration that the instrument can be used in.

interface - an instrument interface is what allows the controls of the instrument to be changed.

Learning the Instrument

The first step that must be taken in writing an HP ID is learning the operation of the instrument. Both the front panel of the instrument and the HP-IB command set need to be mastered. The HP-IB operation of the instrument will influence the component section of the HP ID because the HP-IB command set provides the HP ID communication with the instrument. In addition, the front panel operation of the instrument can help with the lay out the panel section of the HP ID.

One good way to learn the instrument's HP-IB programming language is to write BASIC programs that do certain tasks for which the instrument can be used. For example, write a program for an oscilloscope using the autoscale feature to set up a trace on the scope screen, asking for measurements, etc. The advantage of learning the instrument in this way is exposure to the details of the HP-IB command set and, once the HP ID is done, reusing the development environment to write the same programs. The HP-IB traffic generated by the development product should be comparable to the code you originally wrote, providing an easy way to measure the efficiency of the HP ID.

Pay special attention to the details while learning the instrument's operation. It's better to find out what will work and what won't work now rather than later in the middle of testing. For example, some features of the instrument may be impossible to use over the HP-IB and some HP-IB features may not reside on the front panel anywhere. The function of some of the commands over the bus may not be the same as the same features on the front panel. You need to know whether instruments with a menu-driven front panel and/or softkeys have an HP-IB command set that reflects the same menu structure.

Some commands are valid only at certain times. Consider these factors before writing the HP ID to minimize the time in HP ID coding and testing.

The next seven sections summarize the procedure that should be followed to learn the instrument and effectively anticipate HP ID design problems:

1. Identify the controls of the instrument

This is usually an easy task to do from the front panel, however, sometimes it can be tricky matching HP-IB commands to instrument controls. The best approach is to assign the controls of the instrument (frequency, controls, voltage controls, etc.) to HP-IB commands and queries. Therefore, controls for which there are no HP-IB commands or queries are impossible to implement into an HP ID because there is no way for the HP ID to interact with the instrument to SET or GET the value of a certain control.

Usually, every control on the instrument will correspond to one HP-IB command, possibly with numbers or parameters for each of the possible settings. For example, on a function generator, AM0 and AM1 turn amplitude modulation off and on. The commands FN0 through FN4 may set the type of function (sine, square, etc.) to be generated. This should suggest to the HP ID writer that there are two controls: one for amplitude modulation with two possible settings, and one for function with five possible settings. The HP-IB commands AM and FN are the interface to the instrument controls for amplitude modulation and function type.

Scanning the instrument's command set won't always indicate which commands relate to separate controls on the instrument. On the HP 54502/3A Digitizing Oscilloscopes, there is a control that turns channel 1 off or on. However, over the interface bus, the command VIEW CHAN1 turns the channel on and the command BLANK CHAN1 turns the channel off. Although the commands may appear different, the functionality is the same, and the HP-IB commands VIEW and BLANK are actually interfaces to the same control on the instrument.

2. Identify the menu structure of the instrument, if there is one

A menu structure in the HP-IB command set is usually easy to spot. When a certain command must be prefixed with a keyword, or if the manual says the command is only valid in a certain menu, there is a menu structure in the HP-IB command set. Once the menu structure has been identified, it sometimes helps to write it out. Following is an example of a menu structure from the HP 54111D digitizing oscilloscope:

```
Main level
 Chan1
 Chan2
 Timebase
 Trigger
  Edge
  Pattern
  State
  Time
  Events
 Delta-V
 ...
```

In each menu, HP-IB commands exist that are not valid, or work differently, in a different menu.

Always include any menu path with the HP-IB command. Then the command will work no matter when it is sent to the instrument. For example, to change channel 1 sensitivity, above, the command CHAN1 SENS must be used instead of just SENS. If just SENS was used and the oscilloscope happened to be in the timebase menu, the time sensitivity would be changed instead.

Most of the simple and moderately complex instruments do not have a menu structure. The more complex instruments, however, usually do.

3. Isolate the various modes of the instrument

An instrument mode holds the value for the current setting of its present function. For example, in a digitizing oscilloscope, the user has a choice of triggering in edge, pattern, or state mode. In edge trigger mode, there is a choice of triggering on the rising or falling edge of the signal. However, in the pattern trigger mode, the oscilloscope will trigger whenever the logic levels of the channels match the specified pattern, for example, when channel 1 is high and channel 2 is low. The current setting of the edge control makes no difference in the operation of the oscilloscope when set in the pattern trigger mode. Therefore, both edge and pattern qualify as modes of the oscilloscope. When either edge mode or pattern mode is selected, an additional control is necessary to determine the state of the whole instrument. The additional control also has no impact on the operation of the instrument when the edge or pattern mode is not selected.

Instruments with a menu structure in their HP-IB command set often require that the instrument be in the correct menu in order to send any commands in that menu to the instrument. Usually, however, the settings of the controls in a certain menu will still affect the operation of the instrument when a different menu is selected. Since the controls still have an affect, each menu choice does not qualify as a separate mode of the instrument, because the controls under each menu are still valid in other menus, even though the HP-IB commands are not.

A different way to look at the difference between instrument menus and instrument modes is as follows: instrument menus refer to whether HP-IB commands are valid or not, whereas instrument modes indicate whether the control itself is valid, that is, whether the control on the instrument will have any impact on the operation of the instrument. An instrument mode will frequently have an instrument menu associated with it. When the control is not valid on the instrument, the corresponding HP-IB command will also not be valid. On the other hand, not all instrument menus will have an instrument mode associated with them.

Instrument modes present a special concern to the HP ID writer. The controls under a certain mode are not valid in other modes. To change the value of one of these controls, the mode must be changed first. Changing a menu at any time will not affect the operation of the instrument. However, since this is a mode, the actual operation of the instrument will change when a control under it is changed. To make matters more complex, during the run-time environment, the user could unknowingly change the mode of operation of the instrument by changing a control under the mode. The HP ID needs to take this into account (see "Developing SET ACTION and GET ACTION" later in this chapter).

4. Identify the type of each control

Most controls in the instrument will be integers, reals, or discrete (a set of choices). Other types, such as strings or arrays, are not so common. Decide what type each control in the instrument is.

5. Identify the range or values of each control

Every control on the instrument should have an absolute maximum and an absolute minimum value. Determine these values before writing the HP ID to reduce problems later. Keep in mind that sometimes the maximum and minimum values of a control will change depending on the settings of other controls on the instrument. A **soft range** is a range that “floats” according to the settings of other parameters. For example, the HP 54111D HP ID has a probe adjustment control. For its initial setting of 1:1 probe adjustment, the channel sensitivity will have a range of 1 mV to 5 V. However, with a probe adjustment of 10:1, the sensitivity will range from 10 mV to 50 V. Therefore, the range of values for channel sensitivity depends on the setting of the probe adjustment control. The HP ID was written to accommodate the absolute minimum and maximum values for the channel sensitivity control.

It is possible to use soft ranges in the HP ID so that the correct maximum and minimum values are reflected in the HP ID at all times. To use a soft range this way, the HP ID writer must spend some time trying to figure out the algorithm that is used by the instrument to determine the ranges for each control. For example, the channel sensitivity range in the HP 54111D is directly proportional to the probe adjustment setting and inversely proportional to the number of screens being displayed. The implementation of these algorithms in the actual HP ID can be complex and sometimes lead to conflicts with other components due to couplings and GETs that must be added. The use of soft ranges is time-consuming and adds extra code to the HP ID. For complex HP IDs, it is easier to use absolute limits with verification in the PANEL SET ACTIONS. An alternative is to couple the components that hold the soft limits to any component that may change the range of allowable values.

For discrete controls, there will be a set of choices for which the control can be set instead of a range of choices. In some cases, the available discrete choices can change depending on the settings of other components in the HP ID. The values list should include every possible choice for the control. If certain values are not valid at all times the disable/enable command is used to turn off/on discrete options at appropriate times.

6. Identify the initial value of each control

Every control on the instrument needs to have an initial value. The initial value on powerup is sometimes not correct if the instrument can remember its old setting when it is turned off. Most instruments usually have a reset button or an HP-IB command that will reset the instrument. The initial value will then be the value of the control when the entire instrument is reset.

In rare cases, an instrument may have a control "survive" an instrument reset and turn the instrument off. Such controls present special headaches if found too late in the writing process. Take time to reset the instrument, change the value of the control, and reset the instrument again *FOR EACH CONTROL*! If any controls have different values after the two resets they are of the survive-a-reset type. Components of this type will have no initial value and you will have to work around this problem when writing the HP ID. Generally, you will need to mark each such component as NOPOKEINITIAL, and possibly do a GET of each such component at the end of the INITIALIZE COMPONENT's SET ACTIONS.

7. Summarize the HP-IB command set

Every control that you decide to include in the HP ID will have an HP-IB command associated with it. Take the time to summarize the command set in the following form for easy reference. The following example illustrates an oscilloscope command set:

```
Control: Timebase sensitivity
HP-IB command: TIME:SENS <real number>
 command?  Y query?  Y
Valid menu: Timebase menu (Notice the TIME prefix in HP-IB command)
Valid mode: None - the control is always valid
       Type: Real number
Range of values: 100 ps to 5 s
Initial value: 1 us
Survive a reset? No
Control: Trigger pattern logic, channel 1
HP-IB command: TRIG:MODE:PATTERN:LOGIC { LOW | HIGH | DONTCARE }
 command?  Y query?  Y
Valid menu: Trigger/Pattern menu (pattern menu is a sub-menu of the
      trigger menu)
Valid mode: Trigger/Pattern (control, not just the HP-IB command, is
               valid only in pattern trigger mode)
Type: Discrete
Range of values: LOW, HIGH, DONTCARE
Initial value: LOW
Survive a reset? No
```

Note

"Valid menu" is the menu, if any, that the instrument must be in for the *command* to be valid.

"Valid mode" is the mode, if any, that the instrument must be in for the *control* to be valid.

Coding the Driver

The component section and the panel sections of the HP ID are usually developed in parallel, that is, as another component is added, the panel widget that goes with that component is added, too.

It helps while writing the HP ID code to periodically check the code with the HP ID compiler. This is a tool that locates errors in the HP ID.

1. Prepare a skeleton component outline

Recall that every component in the HP ID must have a name and a type. It is good practice to include an initial value. Use the information in the previous table and assign exactly one component to each control to quickly create a skeleton HP ID. Usually it is helpful to name each component something similar to the control it represents, and to group components in the same menus and modes together. The HP ID will have many components that look like the following component:

```
COMPONENT FOO;
  TYPE <Foo_type>;
  VALUES <Foo_values>;
  INITIAL <Foo_initial_value>;
END COMPONENT;
```

Note that component type determines the values statements syntax.

2. Prepare a skeleton panel outline

After a new component is added, the corresponding panel widget should also be added in the panel section. The five different panel widgets: continuous, discrete, toggle, button and input, are directly related to the component type. The following table assigns panel widget types to component types:

Table 5-1.

Component type	Widget type
INTEGER	CONTINUOUS, INPUT
CONTINUOUS	CONTINUOUS, INPUT
DISCRETE	DISCRETE,TOGGLE (for binary components)
STRING	INPUT
RARRAY	TRACE on an XY
IARRAY	TRACE on an XY

A BUTTON widget is usually assigned to an INTEGER component, but no data is entered with the button widget. When the user clicks on the button, the value of the component is set to 1 and the SET ACTIONS are executed.

RARRAY and IARRAY components are usually used only to receive data from the instrument, and so do not have an input widget for the panel section. To allow an RARRAY or IARRAY component to be read from the instrument, a dummy integer component can be created with the following SET ACTIONS:

```
COMPONENT DATA_ARRAY;
  TYPE IARRAY <size>;
  INITIAL INVALID;
  GET ACTIONS
    ...
  SET ACTIONS
END COMPONENT;
```

(Continued)

```
COMPONENT DUMMY_COMPONENT;
  TYPE INTEGER;
  INITIAL INVALID;
  SET ACTIONS;
    GET DATA_ARRAY;
  END ACTIONS;
END COMPONENT;
```

Then the DUMMY_COMPONENT can be assigned to a button on the panel of the HP ID. When the user clicks on the panel button, the development environment will execute the SET ACTIONS of the DUMMY_COMPONENT, which will get the array from the instrument.

The panel section of the HP ID consists of panels nested inside one another. Each widget that is added to the HP ID will have to be inside one of these panels. An easy way to design the panel structure of the HP ID is to follow the menu structure of the instrument's HP-IB command set. This information will be at your fingertips if the table described in the previous section was completed. A good procedure in developing the panel section is:

1. Declare the panels first, naming each panel similar to the menu/mode it represents to prevent confusion. For an instrument that has two different menus that contain voltage and frequency controls, the panel structure would then resemble the following:

   ```
   PANEL ROOT;
     PANEL VOLTAGE_SETTINGS;
     END PANEL;
     PANEL FREQUENCY_SETTINGS;
     END PANEL;
   END PANEL;
   ```

   The root panel must always be included in any HP ID, so for an instrument with no menu structure only the root panel is necessary.

2. Add the widgets, one corresponding to each component, to the appropriate panel or subpanel. If the HP-IB command associated with a control is only valid in a certain menu, then the widget associated with the component that contains that HP-IB command will be put in the corresponding panel.

   Keep in mind that, once you have subpanels in the HP ID, you must add code in the RESET COMPONENT to SHOW or HIDE the appropriate panels. Only the root panel is automatically shown. The use of a menu component with SHOWs and HIDEs can be useful, also, as shown in the following HP ID:

```
COMPONENT RESET NOTSAVED NOGEN;
  TYPE INTEGER;
  INITIAL DONTCARE;
  SET ACTIONS;
    CLEAR;
    OUTPUT STRING "RESET;";
    FLUSH;
    POKEINITIAL;
  END ACTIONS;
  PANEL SET ACTIONS;
    SET MENU;
  END ACTIONS;

COMPONENT MENU;
  TYPE DISCRETE;
  VALUES CHAN1, CHAN2, TIMEBASE;
  PANEL SET ACTIONS;
    HIDE CHAN1_PANEL;
    HIDE CHAN2_PANEL;
    HIDE TIME_PANEL;
    SELECT MENU;
      CASE CHAN1;
        SHOW CHAN1_PANEL;
      CASE CHAN2;
        SHOW CHAN2_PANEL;
      CASE TIME;
        SHOW TIME_PANEL;
      CASE ELSE;
        SHOW CHAN1_PANEL;
    END SELECT;
  END ACTIONS;
END COMPONENT;
```

(Continued)

```
PANEL MAIN_PANEL;
  POSITION 1,1;
  SIZE 214,213;

  PANEL Top1_panel;
    POSITION 0,182;
    SIZE 214,31;
    FOREGROUND (255,255,255),100;

    BUTTON Reset;
      POSITION 3,6;
      SIZE 50,19;
      LABEL "Reset";
      BACKGROUND (0,130,70),0;
    END BUTTON;

    DISCRETE Menu;
      POSITION 88,6;
      BACKGROUND(75,0,240),0;
      SIZE 121,19;
      LABEL "Channel 1", "Channel 2", "Timebase";
    END DISCRETE;
  END PANEL;

  PANEL CHAN1_PANEL;
    POSITION 1,1;
    SIZE 212,180;
  END PANEL;

  PANEL CHAN2_PANEL;
    POSITION 1,1;
    SIZE 212,180;
  END PANEL;

  PANEL TIME_PANEL;
    POSITION 1,1;
    SIZE 212,180;
  END PANEL;

END PANEL;
```

The main menu component uses UPDATE ACTIONS as the means for accomplishing the appropriate SHOWs and HIDEs every time the component is updated in panel mode only. The action list should first HIDE all the subpanels, then SHOW the appropriate one. If this method is not used, the panels may flash every time the main menu component is changed.

3. Developing SET ACTION and GET ACTION

For every component in an HP ID that corresponds to a control on the instrument (excluding dummy components or other special components) there should also be both SET and GET ACTIONS whenever possible. Sometimes, however, the GET ACTIONS cannot be programmed because there is not an HP-IB command to query the instrument. The GET ACTIONS should always be included if there is an instrument query available. Without any GET ACTIONS, it becomes impossible to automate the testing of the HP ID or make a live query of the instrument value. Thus, any testing must be done manually, which is time-consuming and not very accurate.

Every SET/GET ACTION list that sends a command to, or queries a value from, the instrument must use the OUTPUT and ENTER statements (see “ENTER” and “OUTPUT” in chapter 8, “Component/Action Syntax”). In addition, some actions will have to include modifications to account for instrument modes and menus.

The first step in writing the SET/GET ACTIONS of a component is to implement the HP-IB command/query in the component. For real or integer components, the syntax is simple. For example:

```
SET ACTIONS;
  OUTPUT Component_name FORMAT ’"<HP-IB command> ",K,";"’;
END ACTIONS;

GET ACTIONS;
  OUTPUT STRING "<HP-IB query>";
  ENTER Component_name FORMAT K;
END ACTIONS;
```

For discrete components, the following structure can be used for the SET/GET ACTIONS:

```
SET ACTIONS;
  OUTPUT STRING "<HP-IB command> ";
  OUTPUT Component_name TABLE 'Choice1;','Choice2;', ... ,'ChoiceN;';
END ACTIONS;

GET ACTIONS;
  OUTPUT STRING "<HP-IB query>";
  ENTER CHARSTRING FORMAT "AAAA....";  ! Enter String
  SELECT CHARSTRING;
  CASE "<String1>";
    FETCH (Component_name)Value1;
  CASE "<String2>";
    FETCH (Component_name)Value2;
    ...
  CASE "<StringN>";
    FETCH (Component_name)ValueN;
  CASE ELSE;
    FETCH Component_name;
  END SELECT;
  STORE Component_name;
END ACTIONS;
```

The previous GET ACTIONS use the dummy component "charstring" to read a string from the instrument, then compare charstring to the different possible instrument responses to fetch the correct discrete value for Component_name onto the stack. Next, the value is stored into the component by the STORE statement. Remember, when using string comparisons with a development product, the strings *must* be of equal length or the comparison is automatically false. Therefore, the length of String1, String2, ... , StringN in the select statement must match the number of As in the enter statement.

The same charstring component can be used for all the GET ACTIONS for all the components. Its form is the following:

```
COMPONENT CHARSTRING NOTSAVED NOGEN NOERRCHECK;
    TYPE STRING 255;
    INITIAL INVALID;
END COMPONENT;
```

The CASE ELSE statement is necessary in the GET ACTIONS of the discrete component to prevent an error message from happening in non-live mode. In non-live mode, the output and enter statements in the GET ACTIONS will not be executed, but the select statement will be. The contents of component charstring will not be valid and, if the CASE ELSE statement is not there, the development product will issue an error message. It does this when there is not a match for the value on the stack in the CASE selections in the construct, which for a non-live mode query is a null string or a 0.

The SET/GET ACTIONS for IARRAY or RARRAY components are not as simple because the exact format of output statements and enter statements depends on the application and the instrument. See the system documentation for details on reading in array components.

The <HP-IB command> in the above examples refers to the HP-IB command that is sent to the instrument. The tables you created summarizing the HP-IB command set will have the appropriate entry. Note that any prefix to the command that selects the appropriate menu for the command to be valid should be included in <HP-IB command>. This guarantees that the component can be used regardless of what the current menu setting in the instrument is.

The last step in completing the SET/GET ACTIONS is to allow for any instrument modes that the instrument may have. One problem with instrument modes is difficulty returning them to their original mode when they have been changed to a second mode. For example, suppose the instrument is currently operating in a certain mode and you want to SET or GET a component/control in a different mode. Part of the HP-IB command string that the component sends out must include a prefix that places the instrument in the correct mode that contains the control you want to SET/GET. Once the control has been SET or queried, the instrument remains in the new mode of operation. You can't output an HP-IB command that returns the instrument to its original mode, because you don't know what the original mode was. The solution is to execute the SET ACTIONS of the mode component that contains information regarding the original mode of operation. Now you have resolved the problem: the development environment has a correct state, the component has been SET/queried, and the instrument is still in the original mode of operation.

An example of proper mode manipulation from the HP 54111D HP ID is:

```
COMPONENT TRIGGER_MODE;
  TYPE DISCRETE;
  VALUES EDGE,PATTERN,STATE,TIME,EVENTS;
  INITIAL EDGE;
  SET ACTIONS;
    OUTPUT STRING "TRIG MODE ";
    OUTPUT TRIGGER_MODE TABLE ’EDGE;’,’PAT;’,’STAT;’,’TDLY;’,’EDLY;’;
  END ACTIONS;
  GET ACTIONS;
    OUTPUT STRING "TRIG MODE?";
    ENTER CHARSTRING FORMAT "AAAA";
    SELECT CHARSTRING;
    CASE "EDGE";
      FETCH (TRIGGER_MODE)EDGE;
    CASE "PAT ";
      FETCH (TRIGGER_MODE)PATTERN;
    CASE "STAT";
      FETCH (TRIGGER_MODE)STATE;
    CASE "TDLY";
      FETCH (TRIGGER_MODE)TIME;
    CASE "EDLY";
      FETCH (TRIGGER_MODE)EVENTS;
    CASE ELSE;
      FETCH TRIGGER_MODE;
    END SELECT;
    STORE TRIGGER_MODE;
  END ACTIONS;
END COMPONENT;
COMPONENT EDGE_TRIGGER_SOURCE;
  TYPE DISCRETE;
  VALUES CH1,CH2,TRIG3,TRIG4;
  INITIAL CH1;
  SET ACTIONS;
    ! Note that MODE EDGE must be included in the command to allow state
    ! recalls without learn string...
    OUTPUT STRING "TRIG MODE EDGE SOUR ";  ! Set to edge mode
    OUTPUT EDGE_TRIGGER_SOURCE TABLE ’CHAN1;’,’CHAN2;’,’TRIG3;’,’TRIG4;’;
    SET TRIGGER_MODE; ! Set it back to original mode
  END ACTIONS;
```

(Continued)

```
  GET ACTIONS;
    OUTPUT STRING "TRIG MODE EDGE SOUR?";
    ENTER CHARSTRING FORMAT "AAAAAAAAAAAA";
    SELECT CHARSTRING;
    CASE "CHAN       1";
      FETCH (EDGE_TRIGGER_SOURCE)CH1;
    CASE "CHAN       2";
      FETCH (EDGE_TRIGGER_SOURCE)CH2;
    CASE "TRIG       3";
      FETCH (EDGE_TRIGGER_SOURCE)TRIG3;
    CASE "TRIG       4";
      FETCH (EDGE_TRIGGER_SOURCE)TRIG4;
    CASE ELSE;
      FETCH EDGE_TRIGGER_SOURCE;
    END SELECT;
    STORE EDGE_TRIGGER_SOURCE;
    SET TRIGGER_MODE; ! Restore previous mode of operation
  END ACTIONS;
END COMPONENT;
```

The mode component is TRIGGER_MODE, which allows a choice of operating modes of the oscilloscope. The EDGE_TRIGGER_SOURCE component corresponds to a control on the oscilloscope that is only valid in the edge trigger mode of operation. Therefore, the HP-IB command for EDGE_TRIGGER_SOURCE includes the prefix TRIG MODE EDGE, which changes the instrument's current menu to trigger and the current mode to edge. EDGE_TRIGGER_SOURCE can now be SET or queried, but the instrument is now in the EDGE_TRIGGER mode of operation. The last line of both the SET and the GET actions, SET TRIGGER_MODE, executes the SET ACTIONS of the TRIGGER_MODE component to restore the instrument to the original mode of operation. Note that this method will work no matter what the original mode of operation was, but if the original mode of operation was EDGE, then there will be some extra HP-IB code generated.

The summary of HP-IB commands that you created while learning the instrument should make it easy to find the components that must be modified to account for modes of the instrument. The components can be altered in exactly the same manner as illustrated. Instruments such as the HP 54111D have modes that are nested two deep. For example, in the HP 54111D, the trigger level control depends on which source is selected. Therefore, to SET/query the TRIGGER_LEVEL_CH1 component, first, select the edge mode of operation. Next, SET the EDGE_TRIGGER_SOURCE to channel 1 to make the TRIGGER_LEVEL_CH1 control valid on the instrument.

```
COMPONENT TRIGGER_LEVEL_CH1;
  TYPE CONTINUOUS;
  VALUES RANGE -160E3,160E3,1E-6;
  INITIAL 0.0;
  SET ACTIONS;
    OUTPUT TRIGGER_LEVEL_CH1 FORMAT '"TRIG MODE EDGE SOUR CHAN1 LEV ",K,";"';
    SET EDGE_TRIGGER_SOURCE;
  END ACTIONS;
  GET ACTIONS;
    OUTPUT STRING "TRIG MODE EDGE SOUR CHAN1 LEV?";
    ENTER TRIGGER_LEVEL_CH1 FORMAT K;
    SET EDGE_TRIGGER_SOURCE;
  END ACTIONS;
END COMPONENT;
```

Notice that the prefix to the HP-IB command is now TRIG_MODE_EDGE_SOUR_CHAN1; this prefix sets the appropriate menus/modes. To restore the original modes, the SET ACTIONS for EDGE_TRIGGER_SOURCE are executed, which executes the SET ACTIONS for TRIGGER_MODE, as just discussed.

This completes the basic HP ID. Undoubtedly, there will be situations that will call for innovation that no manual can cover completely. The best way to handle problems is trial-and-error, that is, to experiment with different solutions and choose the best one.

While it is possible to add all the components at once, it is advisable to add them one at a time, then read the unfinished HP ID into the development environment to see whether the code actually works. This will save time in debugging an HP ID at a later time.

4. Finishing touches

Internal components

Components in an HP ID can be used as variables to store a number or value in much the same way as variables in a programming language. Such components will only have a name with no SET or GET ACTIONS. Using components for variable storage is a perfectly legal thing to do, then other components in the HP ID can reference the value stored in the component. Component variable storage can be used as a method of parameter passing between two components. Component variable storage can also be used for other functions such as remembering the last value of components or as a simple placeholder in a SELECT ... CASE ... ELSE construct, a construct that requires that a component name be specified. Since these internal components are only used as variables, always use them with the NOTSAVED NOGEN NOERRCHECK options to save time and space in the HP ID.

Valid Reset State

A working HP ID must have a valid state at all times, that is, the development product's values in the components must match the instrument's control settings. To make this match, the development product tracks the state of the instrument using the HP ID as a guide of how to match the values and settings. There needs to be a state that guarantees that the states of the HP ID and instrument are exactly the same. This is known as a "reset state."

The reset component specified by the INITIALIZE COMPONENT statement in the HP ID will be the component that configures the HP ID in a valid reset state. To create this configuration, both the control values on the instrument and the component values in the HP ID must be set to the same initial value. The instrument will have a reset command that sets all the controls to their initial value, unless they are of the survive-a-reset type. Every component in the HP ID should have an initial value matching the corresponding initial value of the instrument control. All the INITIALIZE COMPONENT needs to do, provided that no controls on the instrument survive a reset, is send the instrument the reset HP-IB command, then initialize the HP ID values with the POKEINITIAL statement.

When one or more components/controls are known to survive a reset, the INITIALIZE COMPONENT should perform a GET on the component. This will guarantee that the reset state is always valid, although it will not be constant for the components that survive a reset. A GET, instead of a SET, is used for two reasons:

1. Often, SETs will involve couplings or other side effects, while GETs are usually "clean," that is, they do not alter the values of other controls of the instrument.
2. Using a GET preserves the functionality of the instrument. That is, after a reset, the HP ID will reflect a natural state of the instrument; the instrument state does not necessarily reflect the state of the HP ID. The entire purpose of the HP ID is to allow the development environment to track the state of the instrument by "understanding" the operation of the instrument.

Error Checking in the Driver

Most instruments have an HP-IB command that asks if there have been any errors. The instrument responds with the appropriate answer. If error checking is present, the error component can be used for automatic error checking in the HP ID. The ERROR COMPONENT statement at the top of the HP ID declares which component in the HP ID is the error component.

When the error checking mode is active, each time there is a SET or a GET on any component that component will execute the GET ACTIONS of the error component *unless* the NOERRCHECK option is set for that component or SKIPERRORCHECK is done in the action list (see "NOERRCHECK" in chapter 8, "Component/Action Syntax"). If, after the GET ACTIONS have been executed, the error component's value is zero, the development product will take no other action. Otherwise, the component's number is taken as the error number and notifies the user that there has been an error.

If the instrument has an error queue, the error component should take steps to make sure that the queue is empty at the end of every error check. For example, using the error queue of the HP 54501A, the HP ID is as follows:

```
COMPONENT ERR_NUMBER NOTSAVED NOERRCHECK;
  TYPE INTEGER;
  INITIAL 0;
  GET ACTIONS;
    OUTPUT STRING ":SYSTEM:ERR?;";
    ENTER ERR_NUMBER FORMAT K;
    SELECT ERR_NUMBER;
    CASE 0;
      ! no error ...
    CASE ELSE;
      GOSUB FLUSH_ERR_QUEUE;
    END SELECT;
  END ACTIONS;
END COMPONENT;
```

For this instrument, if a zero is returned, there has been no error and the GET ACTIONS are exited. If a non-zero number is returned, there has been an error, so the component's value is non-zero and the action list FLUSH_ERR_QUEUE is executed to ensure that the instrument's error queue is empty.

6

Advanced Topics

Overview

The topics in this chapter vary widely. Although they deal with concepts that are not required to create HP IDs, they can make your HP ID more useful and powerful.

Initializing the Driver

The three statements INITIAL, POKEINITIAL, and INITIALIZE, work together to provide HP ID initialization. The HP ID is initialized when:

- The HP ID is added to a soft test system in the development environment.
- The subprogram hpt_init is executed.
- Its address is changed in the development environment.

When the HP ID is initialized, the development product executes the SET ACTIONS (and PANEL SET ACTIONS in the development environment) of the component specified in the INITIALIZE statement. The SET ACTIONS of this component usually include a POKEINITIAL statement, which sets each component to the value and status specified in its INITIAL statement. In the SET ACTIONS, you should also include the commands you want the run-time environment to send the instrument to put it into the proper state.

Usually the PANEL SET ACTIONS of this component also HIDE all panels except the main panel, then SHOWs the main panel.

In summary, here's what you need to do:

1. Determine which components you want as part of the initialization.
2. Use an INITIAL statement in each component you want to set during initialization. For DISCRETE components, this statement is only valid if you have also used a VALUES statement in the component description.
3. Create a component description similar to the RESET component in the program below. It provides a SET ACTIONS list that
   a. sends the commands to the instrument for initialization.
   b. HIDEs and SHOWs appropriate panels.
   c. includes a POKEINITIAL statement.
   d. the POKEINITIAL statement sets all components that. have an INITIAL statement to the value specified.
4. Use the general INITIALIZE statement and specify the name of the component containing the POKEINITIAL statement.

Reset Button

Hewlett-Packard recommends that you provide a reset button on your instrument's soft panel that allows the user to reset the instrument whenever necessary. Hewlett-Packard also recommends that the component specified by the INITIALIZE statement be written so that the instrument is reset regardless of its previous state. Usually this requires sending a device clear (CLEAR) to the instrument. Normally, a device clear resets the instrument parser regardless of its current state. Usually it is beneficial to use some sort of reset syntax (along with device clear) to reset the instrument. Reset syntax/device clear provides a much faster initialization than sending all of the incremental configuration commands.

```
INITIALIZE COMPONENT RESET;
!
COMPONENT RESET;
     TYPE INTEGER;
     SET ACTIONS;
          CLEAR;
          OUTPUT STRING "*RST";
          POKEINITIAL;
     END ACTIONS;
     PANEL SET ACTIONS;
          HIDE Other;
          HIDE About;
          SHOW Main;
     END ACTIONS;
END COMPONENT;
```

How Recall Works

When the development product recalls a state, it performs the following steps:

1. The development product updates the value and status of all components in the HP ID to the corresponding values in the state being recalled.
   The development product also creates a list of components whose SET ACTIONS (and PANEL SET ACTIONS in the development environment) are executed by using the following algorithm:
   - If the HP ID component is VALID, then it is compared with the component from the recalled state. If the values are the same, no further action is taken. If the values differ, the run-time environment copies the new value into the HP ID component and the component is added to the list of components requiring updates.
   - If the HP ID component is INVALID or DONTCARE, then the value from the recalled state is copied into the HP ID component and the component is added to the list of components requiring updates.
2. During the development environment, the development product executes the SET ACTIONS of the component that is specified in the RECALL COMPONENT statement. Also, in the development environment, the development product executes the PANEL SET ACTIONS.

3. The development product executes the SET ACTIONS of every component that is in the list generated in step 1. The SET ACTIONS will be executed in the order in which the components are defined in the HP ID.

   If a component is validated during a state recall or while executing the actions of the RECALL COMPONENT, it is removed from the list of SET ACTIONS to be executed. Since the run-time environment goes through the HP ID in the order of the components, validating a component defined earlier in the list will not prevent its action list from being executed.

   If a component is invalidated during a state recall or while executing the actions of the RECALL COMPONENT, it will be added to the list of SET ACTIONS to be executed. Since the development environment goes through the HP ID in the order of the components, invalidating a preceding component will not prevent its action list from being executed.

Using HIT, UPDATE, and PANEL ACTIONS

The development environment includes four special types of action lists that control an HP ID's soft panel. These are:

- HIT ACTIONS
- UPDATE ACTIONS
- PANEL SET ACTIONS
- PANEL GET ACTIONS

Each of these action lists describe some sort of action that is not used during run time.

HIT ACTIONS

HIT ACTIONS are in the PANEL section of the HP ID. They are used to replace the development product's normal behavior when a user clicks on a BUTTON, DISPLAY, or XY. If one of these panel elements includes HIT ACTIONS, the development environment will not execute any other action list.

The development product normally generates an hpt_set or hpt_get (with Log HP ITG Calls mode on), then executes an hpt_set or hpt_get on the component associated with the panel element. Therefore, the HIT ACTIONS must perform any desired code generation and/or any SETs or GETs.

Note

HIT ACTIONS should usually be used for XY displays because XY panel elements tend to be associated with several components. HIT ACTIONS indicate what the development environment should do when the user clicks on the XY panel element.

Be careful using HIT ACTIONS because it can influence the way some products interact with components. Generally, interactions with the instrument should be made through SET/GET actions rather than HIT ACTIONS.

UPDATE ACTIONS

UPDATE ACTIONS provide an alternative to PANEL SET ACTIONS and PANEL GET ACTIONS for keeping the soft front panels *live* in the development environment.

UPDATE ACTIONS are part of a panel element description. They are associated with the same component as the panel element. UPDATE ACTIONS are executed whenever the associated component's value is changed. UPDATE ACTIONS provide a way of distinguishing the visual representation of a component on the panel from the control of the actual instrument (such as the component itself).

Be careful using UPDATE ACTIONS because it can influence the way some products interact with components. Generally, interactions with the instrument should be made through SET/GET actions rather than UPDATE ACTIONS.

Because UPDATE ACTIONS are associated with the component through the panel element, multiple UPDATE ACTIONS may be associated with one component. Normally, multiple UPDATE ACTIONS are used to manage the different representations of the component as provided by the different panel elements.

PANEL SET ACTIONS

In the development environment, PANEL SET ACTIONS are executed after a component's SET ACTIONS are executed. PANEL SET ACTIONS are not executed in the run-time environment. Using PANEL SET ACTIONS and SET ACTIONS provides a way of keeping the soft panel *live* while not impacting performance when your program is running.

PANEL GET ACTIONS

In the development environment, PANEL GET ACTIONS are executed after a component's GET ACTIONS are executed. PANEL GET ACTIONS are not executed in the run-time environment. Using PANEL GET ACTIONS and GET ACTIONS provides a way of keeping the soft panel *live* while not impacting performance when your program is running.

Summary

HIT ACTIONS are used to alter the default behavior of the development environment when the user clicks on a DISPLAY, BUTTON, or XY. HIT ACTIONS are largely independent of the other action lists.

UPDATE ACTIONS and PANEL SET/GET ACTIONS provide alternative ways of including actions to keep the soft panel *live* while not affecting run-time performance. UPDATE ACTIONS allow the user to update the representation of a component on the soft panel as the component's value or status changes.

PANEL SET/GET ACTIONS allow you to specify operations performed only when running in the development environment. PANEL SET/GET ACTIONS can be used in addition to normal SET/GET ACTIONS. Note that UPDATE ACTIONS are executed when a component's value or status is changed and the value is currently displayed in the development environment, but PANEL SET/GET ACTIONS are only executed when a component's SET/GET ACTIONS are executed.

Using Learn Mode

Some instruments provide what is commonly known as a **learn string**. A learn string is a string that is generated by the instrument that, when sent back to the instrument, returns the instrument to the state it was in when it generated the learn string. For some instruments, the learn string is accepted and acted upon in less time than just a few conventional instructions. For such instruments, it may be desirable to send the learn string during a state recall instead of using the development product's incremental state programming.

To program an instrument using learn strings with the development product, you must extract the string from the instrument when the product stores a state and you must send it back when the product recalls the state. You do this by using a STORE COMPONENT statement and a RECALL COMPONENT statement.

The development product executes the SET ACTIONS of the component specified in the STORE COMPONENT statement whenever the user stores a state. These actions should include the commands needed to extract the learn string from the instrument, placing it in some SAVED component.

As discussed earlier in this chapter in "How Recall Works," the development product executes the SET ACTIONS of the component specified in the RECALL COMPONENT statement after it sets the components to their target values, but before it executes any of the SET ACTIONS of the components. Hence, you should use the SET ACTIONS of the component specified by the RECALL COMPONENT to send the learn string to the instrument and then use a VALIDATE ALL statement to prevent the product from sending any additional commands to the instrument.

For example, consider the HP 54501A Digitizing Oscilloscope. It accepts a learn string in approximately the same amount of time that it accepts four or five conventional instructions. This makes state programming with a learn string a good choice. To implement the learn string, we created three components:

- LEARN_STRING holds the actual learn string.
- STORE_STATE reads the learn string from the instrument.
- RECALL_STATE sends it back to the instrument.

```
!======================
! LEARN STRING SUPPORT
!======================

COMPONENT LEARN_STRING;
  TYPE IARRAY 512; ! 512 INT’s = 1024 BYTEs
  INITIAL INVALID;
END COMPONENT;

COMPONENT STORE_STATE NOTSAVED NOGEN NOERRCHECK;
  TYPE INTEGER;
  SET ACTIONS;
    !
    ! Binary Block Format:
    !
    ! ’#800001024’ 10-byte header
    ! < data  block > 1024-byte binary data
    ! ’LF’ 1-byte LF terminator
    !
    OUTPUT STRING ":SYSTEM:SETUP?";
    ! Skip 10-byte header
    ! Read 1024-byte learn string
    ENTER LEARN_STRING FORMAT INT16 10 512;
    ! Consume "LF’ terminator
    ENTER CHARSTRING FORMAT "#,A";
  END ACTIONS;
END COMPONENT;

COMPONENT RECALL_STATE NOTSAVED NOGEN NOERRCHECK;
  TYPE INTEGER;
  SET ACTIONS;
    ! Download learn string to instrument
    OUTPUT STRING ":SYSTEM:SETUP #800001024";
    OUTPUT LEARN_STRING INT16 512;
    OUTPUT STRING "";
```

(Continued)

```
    ! Validate status of all components in HP ID
    ! to prevent the normal incremental SETs...
    VALIDATE ALL; ! triggers appropriate UPDATE ACTIONS
    ! Invalidate any components which represent
    ! some form of acquisition or measurement
    INVALIDATE VMARKER1;
    INVALIDATE VMARKER2;
    INVALIDATE TMARKER1;
    INVALIDATE TMARKER2;
    INVALIDATE PULSE_MSMT_VALUE;
    INVALIDATE DELAY_MSMT_VALUE;
  END ACTIONS;
  PANEL SET ACTIONS;
    GOSUB UPDATE_TRACE_DATA; ! Get current trace data
  END ACTIONS;
END COMPONENT;
```

Using Saved Components

As mentioned above, if you use a VALIDATE ALL statement in the SET ACTIONS of the component specified in the RECALL COMPONENT statement, the development product will not execute the SET ACTIONS of the other components. With this in mind, you might wonder why you would design an HP ID with any saved components other than the one holding the learn string.

Note

A saved component is any component not specified as NOTSAVED.

A valid alternative is to have no saved components other than the one containing the learn string. This approach minimizes what the development product must do while recalling a state, and it minimizes the amount of memory required to store a state.

However, in some cases you may want to use some saved components so that when the user recalls a state in the development environment, the development product will set the components to their target values. If none of the components in the HP ID are saved, then none of them would change when the user recalled a state. During run time this is of little importance, but it may be desirable in the development environment.

When to Use Learn Strings

Although many instruments use learn strings, there are some disadvantages to using them. When deciding if your HP ID should use learn strings, you should consider:

- How fast the learn string is compared to incremental programming.
  - □ How fast does the instrument accept a learn string versus how fast it accepts conventional instructions?
  - □ How many components will be sent to the instrument during a typical state recall?
- If you store a state using a learn string, the instrument must be present. Without learn strings, you can create states for an instrument without the instrument present.
- Using learn strings relieves the HP ID of the task of sending each incremental programming step in the correct order during a state recall, which can be a major effort in the design of the HP ID. Including learn strings greatly simplifies the task.

Simulating a Component

It is occasionally useful to represent a component with panel elements of a different type than the component. For example, the HP 438A HP ID uses two buttons to get and store a binary string. In this case, it is not desirable to use either the INPUT or DISPLAY panel elements because the string is rather long and does not need to be visible.

For this case, it is desirable to use two buttons. One button will read the string and one will write the string to the instrument. To create these buttons, we've created the component STATE_STRING. STATE_STRING's GET ACTIONS reads the string and its SET ACTIONS sends the string. The two BUTTONs each include HIT ACTIONS. One HIT ACTIONS list includes a GET STATE_STRING statement. GET STATE_STRING causes the development environment to execute the GET ACTIONS list of the STATE_STRING component.

The other HIT ACTIONS list includes a SET STATE_STRING statement. SET STATE_STRING causes the development environment to execute the SET ACTIONS list of the STATE_STRING component. Each HIT ACTIONS list also includes a CODEGEN (available in HP ITG only) statement so that calls to hpt_get or hpt_set will be logged when the user clicks on the appropriate button with Log HP ITG Calls mode on.

```
TEXT 7,84,"Setup String";
BUTTON STATE_STRING;
     POSITION 120,84;
     LABEL "READ";
     HIT ACTIONS;
          CODEGEN GET,LEARN_STRING;
          GET STATE_STRING;
     END ACTIONS;
END BUTTON;
!
BUTTON STATE_STRING;
     POSITION 165,84;
     LABEL "Send";
     HIT ACTIONS;
          CODEGEN SET,LEARN_STRING,"Value";
          SET STATE_STRING;
     END ACTIONS;
END BUTTON;
```

Component Interactions

In general, instruments change several settings as a result of a single command. For example, when changing the Function on a multimeter, the RANGE may also change. Most instrument interactions fall into one of three categories:

- Lockout parameters.
- Parameters that get dragged.

- Parameters where a convenient value is chosen but not required are functionally coupled.

Each of these types of interactions are rare, but they can be very important. For example, an interaction may only occur when some component is set near its usable limit.

Lockout Parameters

A lockout parameter is a command setting that the instrument does not allow while in certain states. Usually, the disallowed command represents an impossible hardware configuration.

Consider a hypothetical situation where an oscilloscope measures rise time using an internal algorithm. First, this algorithm sets the trigger level at the 10% amplitude of the waveform, then sets a marker at the 90% level. Then, the marker time provides the rise time.

For this example, when the oscilloscope is configured to measure rise time, the user is not allowed to set the trigger level. Under these conditions, trigger level is a lockout parameter.

Instead of introducing lockout parameters, many instruments allow the user to set the parameter but then will not use that parameter when making the measurement. The user-supplied value is then used when the instrument returns to conventional operation.

You may also create a subpanel with the component, then HIDE the subpanel when the lockout condition is selected.

Dragged Parameters

This is the most common type of coupling. A dragged parameter is one that must be changed in order for the instrument to execute an instruction.

For example, consider a multimeter that allows a range of 1, 10, and 100 when measuring DC volts, and a range of 10, 100, and 1000 when measuring resistance. The multimeter must pick a legal value for range when the function is changed. That is, if you are measuring volts on the 1 volt range and change the function to resistance, the multimeter will change the range to a legal value, probably the 10 volt range in this case.

The development product's COUPLED statement helps you deal with this type of coupling. The previous example would look something like this:

```
COMPONENT FUNCTION;
     TYPE DISCRETE;
     VALUES DCV,OHM;
     COUPLED RANGE;
     SET ACTIONS;
          OUTPUT FUNCTION TABLE "DCV","OHM";
     END ACTIONS;
END COMPONENT;
!
COMPONENT RANGE;
     TYPE CONTINUOUS;
     VALUES RANGE 1,1000;
     SET ACTIONS;
          OUTPUT RANGE FORMAT "'RANGE',K";
     END ACTIONS;
     GET ACTIONS;
          OUTPUT STRING "RANGE?";
          ENTER RANGE;
     END ACTIONS;
END COMPONENT;
```

The COUPLED statement indicates to the development product that changing FUNCTION may change RANGE. When FUNCTION is set via the soft panel or with hpt_set, the product acts as follows:

- The development environment executes the GET ACTIONS of RANGE, updating the component to the value selected by the instrument. If Live mode is off, nothing is done.
- The run-time environment INVALIDATEs RANGE. This not only saves the overhead of getting RANGE, but also indicates to the development product that, when recalling a subsequent state, the value of RANGE should be updated.

During a recall state, COUPLED has no affect because the stored state is known to be internally consistent. Therefore, if the stored value for RANGE is consistent with the function and the stored value for RANGE is the same as the current one, then the instrument does not need to alter the RANGE setting. Alternatively, if the instrument needs to alter its range setting, the development product will note the change during the state recall and will send the value for RANGE to the instrument. Note that this is not true for some types of component interactions.

For this process to work, it is necessary to send the value for FUNCTION to the instrument before sending the value for RANGE. If RANGE were sent first, it might be inconsistent with the value for FUNCTION and would be ignored by the instrument.

Remember, use COUPLED for the following cases involving dragged parameters:

- When changing one component changes another, that is, leaving it alone would create an illegal state.
- When the changed component has GET ACTIONS.

Functional Couplings

A functional coupling occurs when the instrument chooses a new value for a related component even though the old value was meaningful. For example, consider a source whose output can be disabled. The source may enable its output as a result of a change in the modulation frequency.

Usually one of the more convenient things to do with this sort of coupling is to send the value of the related component to the instrument when the original component changes. In this example, you would send the output enable component to the instrument whenever the modulation frequency is changed. This approach makes the HP ID behave differently from the instrument (that is, changing the modulation frequency does not enable the output).

```
COMPONENT MOD_FREQ;
     TYPE CONTINUOUS;
     SET ACTIONS MOD_FREQ_SET;
END COMPONENT;
!
COMPONENT OUTPUT_ENABLE;
     TYPE DISCRETE;
     VALUES DISABLE,ENABLE;
     SET ACTIONS OUTPUT_ENABLE_SET;
END COMPONENT;
!
```

(Continued)

```
ACTIONS OUTPUT_ENABLE_SET;
     OUTPUT OUTPUT_ENABLE TABLE "OD","OE";
END ACTIONS;
!
ACTIONS MOD_FREQ_SET;
     OUTPUT MOD_FREQ FORMAT "’MF’,K,’HZ’";
     GOSUB OUTPUT_ENABLE_SET;
END ACTIONS;
```

Another alternative is to have the modulation component in the HP ID set its copy of the output enable flag, although this presents a problem when recalling states. To see the problem, consider a stored state with the output disabled. When recalling this state, if the value of the modulation frequency has changed, the SET ACTIONS for modulation frequency will reenable the output, and the state will not be recalled correctly. Therefore, when choosing this alternative, the HP ID should only alter the enable flag while not recalling a state.

```
COMPONENT MOD_FREQ;
     TYPE CONTINUOUS;
     SET ACTIONS MOD_FREQ_SET;
END COMPONENT;
!
COMPONENT OUTPUT_ENABLE;
     TYPE DISCRETE;
     VALUES DISABLE,ENABLE;
     SET ACTIONS OUTPUT_ENABLE_SET;
END COMPONENT;
!
ACTIONS OUTPUT_ENABLE_SET;
     OUTPUT OUTPUT_ENABLE TABLE "OD","OE";
END ACTIONS;
!
ACTIONS MOD_FREQ_SET;
     OUTPUT MOD_FREQ FORMAT "’MF’,K,’HZ’";
     IF RECALLING THEN;
          GOSUB OUTPUT_ENABLE_SET;
     ELSE;
          FETCH (OUTPUT_ENABLE)ENABLE;
          STORE OUTPUT_ENABLE;
     END IF;
END ACTIONS;
```

Tips for State Recall

Without Learn String . . .

The hardest part of writing an HP ID is getting state stores and recalls to work. Debugging a recall problem can create headaches quickly, even for the best HP ID writers. When a state is recalled, the HP ID executes the SET ACTIONS of every component whose value has been changed in the order that the component appears in the HP ID. In other words, during a state recall, you have no control over the order that the component SET ACTIONS are executed other than the order they occur in the HP ID. This order-of-execution constraint is what leads to many of the problems that occur during recall testing.

The key to passing a recall test is to write every component such that its SET/GET ACTIONS will work correctly no matter what menu, mode, or state the instrument is currently in. This was the basic reason for modifying the SET/GET ACTIONS of the components to allow for HP-IB command set menu structures and instrument modes. If an instrument had no command set menu structure, or it was possible to access any control at any time, theoretically, there would be no problems with recall testing other than couplings.

The way to avoid most problems regarding recalling states involves the use of a recall component and a finish recall component. The recall component is a special component with SET ACTIONS that are executed during a state recall before it starts executing any other components. The finish recall component is executed at the end of every state recall. Together, the two components can be used to solve a variety of problems.

The recall component and the finish recall component take the following form:

```
COMPONENT RECALL_STATE NOTSAVED NOGEN NOERRCHECK;
  TYPE INTEGER;
  INITIAL DONTCARE;
  SET ACTIONS;
 ...
    INVALIDATE FINISH_RECALL; ! always executed after a recall
  END ACTIONS;
END COMPONENT;

COMPONENT FINISH_RECALL NOGEN; ! last component in the HP ID
  TYPE INTEGER;
  INITIAL DONTCARE;
  SET ACTIONS;
 ...
  END ACTIONS;
END COMPONENT;
```

When the SET ACTIONS of the recall component are executed before every state recall, the FINISH_RECALL component will automatically be invalidated. The invalidation will tell the development product to execute the SET ACTIONS of the FINISH_RECALL component when it gets to it. Since the components are validated in the order of their placement in the HP ID, the finish recall component should be the last component in the HP ID. Isolating problems with state recalls takes patience, and solving them can at times demand a bit of creativity. It is impossible to describe every type of problem that can occur, but many of them fall into the following categories:

Problem: One component must be set before a second component to avoid side effects, but if the first is placed before the second in the HP ID, it causes an error during parsing due to a forward reference.

Solution 1: In the SET ACTIONS of the recall component, set the component so that it is executed first. This will also validate the component so that it will not be executed later.

Solution 2: Forward references in action lists can be moved to named action lists and then the order of the components can be reversed.

Problem: Many components in the HP ID call the same action list for cosmetic reasons, such as screen updates, but during a recall, this should not happen.

Solution: Use constructs like the IF RECALLING or IF PANELMODE in the action list.

Problem: A component must have its SET ACTIONS executed during every state recall, even if its value does not change.

Solution: Invalidate the component in the recall component. Its SET ACTIONS will be executed when state recall gets to that component.

Problem: A component is changed when a state is stored. When the state is recalled, however, the component's value is different from what is expected. The problem is traced to an order of execution problem, but the problem cannot be solved by positioning in the HP ID or by a SET in the recall component.

Solution: In the finish recall component, do a SET of the component.

Problem: Components A & B each have VALUES RANGE 0,4 but the instrument limits the sum of the two to 6. For other reasons they must be separate components. The states:

```
State 1        State 2
 A  4           A  2
 B  2           B  4
```

will not work for a state recall. If A is before B in the HP ID, then recalling from state 1 to state 2 works, but recalling from state 2 to state 1 causes the <set A to 4> command to be sent while B is still sect to 4, causing an error. Conversely, if B comes before A in the HP ID, recalling from state 1 to state 2 fails.

Solution: In the RECALL COMPONENT do a

```
OUTPUT "<HP-IB command to set A to 0>"
INVALIDATE A
```

With learn string . . .

A learn string is a string of data that can be requested from some instruments; the string contains information about the current state of the instrument. For HP IDs that use learn strings, there will be no problems with state recalling because the HP ID can simply feed the learn string back to the instrument rather than executing the SET ACTIONS list of every component that changes in the new state. This eliminates order of execution problems that could occur if learn strings were not used. To decide whether or not to use learn strings in an HP ID, consider the following concerns:

Advantages of learn strings

1. The time it takes for an instrument to receive a learn string and configure itself to the new state can be much quicker than if learn strings were not used.

2. A learn string HP ID is much easier to code and to test. For a complex instrument this alone may be enough of an argument to include learn strings in the HP ID.

Disadvantages of learn strings

1. Using a learn string makes it mandatory for the instrument to be present to create states. The HP ID is no longer stand-alone in that to save a state or recall a state, the HP ID will need to work with the instrument to create a consistent or valid state. One of the strong points about the HP ID is that for non-learn string HP IDs it is possible to build valid states with no instrument present.

2. For simple instruments a learn string can make recalling slower, rather than faster. Even for complex instruments, if the number of components that change between states is small, learn strings can still be up to three times as slow.

3. In the development environment, after a state recall, many of the component settings will not have been set to the values in the learnstring and will show a different value from that of the instrument. This can be misleading.

If learn strings are a good decision for a particular instrument, use the following procedure to add them to an HP ID:

Note Drivers with learn strings should not do any SETs in the RECALL COMPONENT of any saved components.

1. You must inform the HP ID how to read and store the learn string during state stores and recalls. This is done using store and recall components. The store and recall components can be named anything, but the HP ID needs to mark them as store and recall components. Add the following lines and components to the HP ID:

```
RECALL COMPONENT <recall component name>;
STORE COMPONENT <store component name>;

COMPONENT <recall component name> NOTSAVED NOGEN NOERRCHECK;
  TYPE INTEGER;
END COMPONENT;

COMPONENT <store component name> NOTSAVED NOGEN NOERRCHECK;
  TYPE INTEGER;
END COMPONENT;
```

2. The learn string is actually an array of data that must be saved along with all the other components during a state store. This is accomplished with a dummy learn string component that the store component reads in every time a state is stored. The store components SET ACTIONS will automatically be executed every time a state is stored:

```
COMPONENT LEARN_STRING;
  TYPE IARRAY <learn string length>;
  INITIAL INVALID;
END COMPONENT;
```

3. Add the following SET ACTIONS to the store component:

```
SET ACTIONS;
  OUTPUT STRING <HP-IB learn string query>;
  ENTER LEARN_STRING FORMAT INT16 XXX YYY;
  ENTER CHARSTRING "#,A";  !Eat up the rest of the line
END ACTIONS;
```

XXX is the amount of characters to skip before the data, if any, and YYY is the length of the learn string. The enter charstring line is used to consume any EOL characters that the instrument may send back. Obviously, there must be a string component named "charstring" to make this work.

4. The SET ACTIONS of the recall component are executed each time a state is recalled. Therefore, there needs to be a SET ACTIONS list in the recall component that tells the HP ID how to feed the learn string back to the instrument:

```
SET ACTIONS;
  OUTPUT STRING <HP-IB learn string command>;
  OUTPUT LEARN_STRING INT16 <learn string length>;
   VALIDATE ALL;
END ACTIONS;
```

The VALIDATE ALL statement will keep the HP ID from executing any other SET ACTIONS in the HP ID, as usually happens in a state recall. When a state is recalled, the components that change their value are marked as invalid, then the SET ACTIONS of the recall component are executed. The VALIDATE ALL statement will inform the HP ID that after the learn string has been sent to the instrument, all the components in the HP ID will be valid.

7

Creating Instrument Help

Overview

The development product's online Help system allows you to document the HP ID you have created so that users can get that information quickly.

This chapter describes how you can create online Help for your HP IDs. As a general guideline, we recommend that you use the online Help system within your development product as a quick reference guide to the instrument.

Creating a Help File

Help files must be ASCII files. To create a Help file, you can use the same text editor you used to create the HP ID. The name of the Help file must be the same as the associated HP ID, except the Help file name must end with an .IH extension and meet the MS DOS file-naming requirements.

A Help source file is similar to the component and panel sections of the HP ID in that it consists of a series of HELP ... END HELP statements (see figure 7-1). Each HELP statement requires a topic that the development product uses as a selection in the Help list box for the current instrument (see figure 7-2). Each topic should be less than 40 characters long and each line of the text less than 45 characters. These limitations maintain the required margins for the product's Help window.

7

Following is a Help file example:

```
HELP OVERVIEW
This is a Help file for the HP 3478A.
END HELP
```

(Continued)

```
HELP Using the Panels
This is the second topic of the
file.
END HELP
HELP SRQ Mask
This is the third topic.
END HELP
HELP SRQ Status
This is the fourth topic.
END HELP
HELP Error Register
This is the fifth topic.
END HELP
```

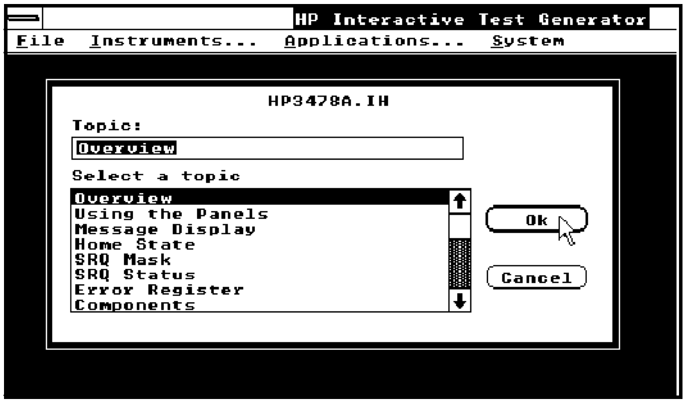


Figure 7-1.
Product Example Uses the Topics as Selections in the List Box

7

Note

When the user adds an HP ID to the product's development system, it looks for a file with a name that matches the HP ID name, except for the difference in file name extensions: `.ID` for the HP ID, and `.IH` for the help file. This means that if you use one Help file for several closely related instruments, you must copy that file so that each instrument has its own Help file with a matching name.

8

Component/Action Syntax

Overview

This section contains an alphabetical reference to the keywords available for use in the component section of an HP ID. Each entry defines the keyword, shows the proper syntax for its use, provides one or more examples, and explains semantic details.

Interpreting the Syntax Drawings

- All characters enclosed in an **oval** must be entered exactly as shown.
- Words enclosed by a **rectangle** are names of items used in the statement.
- *Italic* letters indicate that the word or words are fully explained in their own section of this chapter.
- Statement elements are connected by lines. Each line can be followed in only one direction, with an arrow indicating the direction.

Optional Elements and Their Defaults

An element is optional if there is a path around it. Optional elements usually have default values. The table or text following the drawing specifies the default value that is used when an optional item is not included in a statement.

Naming Rules

- Don't use keywords for component or action names.
- Component and action names can be up to 25 characters long.
- Component and action names must start with an alpha character, A-Z or a-z. That character may be followed by any mix of alphanumeric characters,

either upper or lower case, or by an underscore. The syntax is not case sensitive.

Following is a list of possible component names:

□ FREQUENCY
□ SLOT4
□ SLOT_4
□ Start_Frequency

Comments

A comment may be created by preceding the comment with an exclamation mark. You can also make comments on the same line as a statement. This is done by placing an exclamation mark after the statement.

Spaces, Commas, and Other Separators

In general, a space is required between a keyword and an item. A space or a comma is required between a keyword and multiple items following it.

Note

All component statements must end with a semicolon. More than one action statement may appear on the same line, but they must be separated by semicolons. For example:

```
FETCH 2; STORE V_MAX;
```

ACTIONS

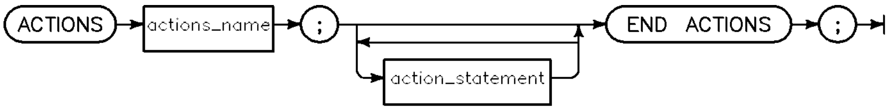
The ACTIONS ... END ACTIONS compound statement allows you to define an action list outside of a component description. Such an action list can be associated with many different components or panel elements, and it may be useful if the action list for one component references another. This type of action list is generally called a *named* action list.

You can call a named action list two ways:

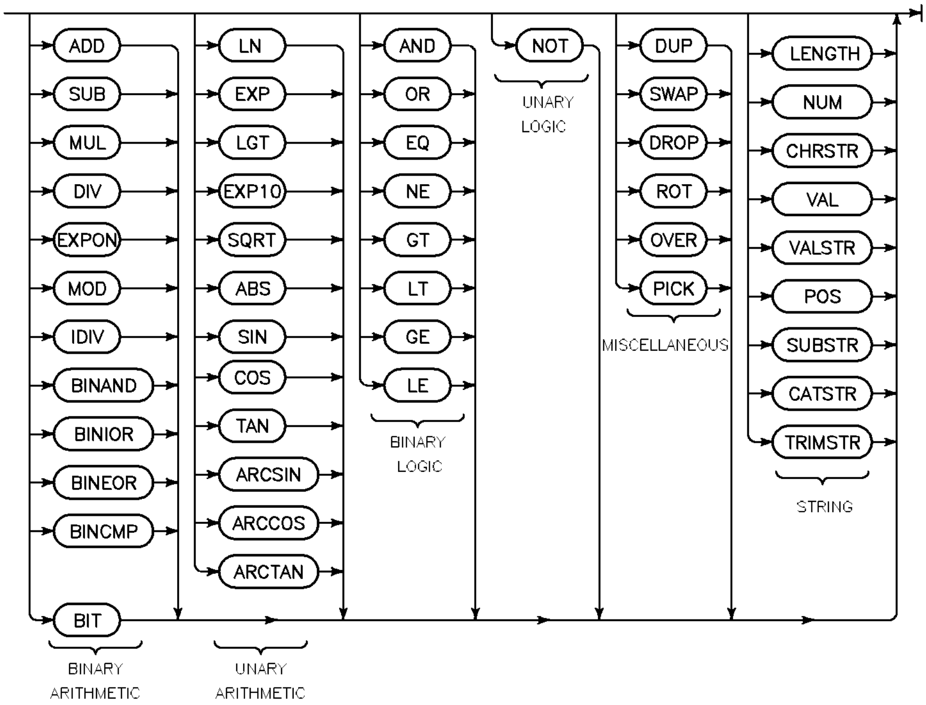
- Specify the name of the general action list in a GOSUB action statement.
- Specify the name of the general action list in a SET ACTIONS, GET ACTIONS, PANEL SET ACTIONS, PANEL GET ACTIONS, HIT ACTIONS, or UPDATE ACTIONS statement.

An action list consists of a series of action statements. The valid statements are shown in tables 8-1 through 8-8. Each statement is fully described in this chapter.

Syntax



ACTIONS



Item	Description
actions_name	The name of the action list.
action_statement	See tables 8-1 through 8-8 for a complete list.

Table 8-1. Action Statements

BITS	GOSUB	SHOW
CLEAR	HIDE	SKIP EOL
CODEGEN	IF	SKIP ERRCHECK
DISABLE	INVALIDATE	SPOLL
DONTCARE	LOOP	STORE
DOWNLOAD	MATSCALE	TRIGGER
ENABLE	NOTIFY	UPLOAD
EXIT IF	OUTPUT	USERSUB
ENTER	POKEINITIAL	VALIDATE
FETCH	SELECT	WAIT SPOLL BIT
FLUSH	SET	WAIT TIME
GET		

Example

```
REVISION 2.0;

ACTIONS FREQ_0;
   FETCH 0;
   STORE FREQ;
END ACTIONS;

COMPONENT DC;
  TYPE INTEGER;
  SET ACTIONS;
    OUTPUT STRING "DC";
    GOSUB FREQ_0;
  END ACTIONS;
END COMPONENT;
```

Arithmetic, Logical, and String Operators

The arithmetic, logic, and string operators (OP) obtain their operands from the stack and return the result to the stack.

- Binary operators operate on the two values on top of the stack as follows:
  - □ second_from_the_top OP top_of_stack
- Unary operators operate on the top stack value.
- Arithmetic operators use the values of the operands and perform REAL arithmetic.
- Logical operators operate on the true (non-zero) and false (zero) values of the operands.
- Bit operators (BINAND, BINIOR, BINEOR, BINCMP, BIT) round their values to 16-bit 2's complement integers before operating on them.
- The miscellaneous operators (table 8-6) DUP, SWAP, DROP, ROT, OVER, and PICK, as well as the following binary logic operators, all work on strings:
  - □ EQ
  - □ NE
  - □ LT
  - □ GT
  - □ LE
  - □ GE
- String operations operate on strings or parts of strings (see tables 8-7 and 8-8).

Note When not specified otherwise, the result of each arithmetic, logical, and string operation is pushed on the stack, replacing whatever values were popped.

To help you understand the order of operation for the binary operators, assume two values, A and B, are fetched, resulting in the following stack configuration:

Example

```
FETCH A;
FETCH B;
      -----------
      |    B    | <-- TOS
      -----------
      |    A    |
      -----------
      |    .    |
      |    .    |
      |    .    |
```

The order of operation is “A operator B.” The original stack configuration is destroyed as the operation result is pushed onto the stack.

Example

```
      -----------
      | result  | <-- TOS
      -----------
      |    .    |
      |    .    |
      |    .    |
```

Note

For the following binary arithmetic operators, the value for B should not equal zero: DIV, EXPON, MOD.

Table 8-2. Binary Arithmetic Operators

Item	Description
ADD	Pops the top two values from the stack and adds them with the result pushed back onto the stack. (A ADD B)
SUB	Pops the top two values from the stack and subtracts the top stack value from the value second to the top of the stack. (A SUB B)
MUL	Pops the top two values from the stack and multiplies them. (A MUL B)
DIV	Pops the top two values from the stack and divides the value second from the top of the stack by the top stack value. (A DIV B)
EXPON	Pops the top two values from the stack and exponentially raises the value second from the top of the stack by the top stack value. (A EXPON B)
MOD	Pops the top two values on the stack and divides the value second from the top of the stack by the top value, and then pushes the remainder of the division on the stack. (A MOD B)
IDIV	Pops the top two values from the stack and divides the value second from the top of the stack by the top stack value, and then pushes the integer portion of the quotient. (A IDIV B)
BINAND	Pops the top two values and performs a bit-by-bit logical AND operation on them after rounding them to 16-bit integers. (A BINAND B)
BINIOR	Pops the top two values and performs a bit-by-bit inclusive OR operation on them after rounding them to 16-bit integers. (A BINIOR B)
BINEOR	Pops the top two values and performs a bit-by-bit exclusive OR operation on them after rounding them to 16-bit integers. (A BINEOR B)
BINCMP	The value at the top of the stack is rounded to a 16-bit integer, and then each bit is complemented. (A BINCMP B)
BIT	Pops the top two stack values, and uses the top stack value to point to a specific bit in the value second from the top of the stack. Zero is the least significant bit. (BIT B of A)

Table 8-3. Unary Arithmetic Operators

Item	Description
LN	Pops the top stack value, and performs the natural log operation on that value.
EXP	Pops the top stack value, and performs the natural log exponentiation operation on that value. Uses the Naperian e value (˜2.718 281 828 459 05) for the exponentiation.
LGT	Performs the base ten logarithm operation on the top stack value.
EXP10	Replaces the top stack value with a value that is 10 to the power of the top stack value.
SQRT	Replaces the top stack value with the square root of the top stack value.
ABS	Replaces top of stack with the absolute value of the top of stack.
SIN	Replaces top of stack with SIN of the top of stack value.
COS	Replaces top of stack with COS of the top of stack value.
TAN	Replaces top of stack with TAN of the top of stack value.
ARCSIN	Replaces top of stack with the inverse SIN of the top of stack.
ARCCOS	Replaces top of stack with the inverse COS of the top of stack.
ARCTAN	Replaces top of stack with the inverse TAN of the top of stack.

Table 8-4. Binary Logic Operators

Item	Description
AND	Pops the top two values, and performs a logical AND operation on them. A non-zero (positive or negative) value is treated as a 1. (A AND B)
OR	Pops the top two values, and performs a logical OR operation on them. A non-zero (positive or negative) value is treated as a 1. (A OR B)
EQ	Pops the top two values and compares them bit-by-bit. If all the bits are the same, then 1 is pushed on the top of the stack. Otherwise, 0 is pushed on the top of the stack. (A EQ B)
NE	Pops the top two values and compares them. If equal, then 0 is pushed on the top of the stack. (A NE B)
GT	Pops the top two values. If the second to the top value is greater than the top value, then 1 is pushed on the stack. Otherwise, 0 is pushed on the top of the stack. (A GT B)
LT	Pops the top two values. If the second to the top value is less than the top value, then 1 is pushed on the stack. Otherwise, 0 is pushed on the top of the stack. (A LT B)
GE	Pops the top two values. If the second to the top value is greater than or equal to the top value, then 1 is pushed on the stack. Otherwise, 0 is pushed on the top of the stack. (A GE B)
LE	Pops the top two values. If the second to the top value is less than or equal to the top value, then 1 is pushed on the stack. Otherwise, 0 is pushed on the top of the stack. (A LE B)

Table 8-5. Unary Logic Operators

Item	Description
NOT	If the top stack value is 0, that value is replaced with a 1. Otherwise, that value is replaced with 0.

Table 8-6. Miscellaneous Operators

Item	Description
DUP	Pushes a duplicate of the top stack value on the top of the stack.
SWAP	Reverses the top two values on the stack.
DROP	Removes the top stack value from the stack.
ROT	Rotates the top 3 items on the stack such that the third value moves to the top and the first and second values move down one place.
OVER	Pushes a duplicate of the second to top stack value on the top of the stack.
PICK	Pops the top value and uses that value as an index into the stack, pushing the indexed value.

Note

All STRING operations are defined for strings only. If numbers are used where strings should be, unpredictable results may occur.

Table 8-7. STRING Operators

STRING	Operators
LENGTH	Pops the top value (string) and returns the length of the string to the stack.
NUM	Pops the top value (string) from the stack and returns the ASCII value of the first character of the string to the top of the stack.
CHRSTR	Takes the value and returns its ASCII character.
VAL	Pops the top value from the stack (string) and converts it to a number.
VALSTR	Pops the top value from the stack (number) and converts it to a string.
POS	Pops two values (strings) from the stack and returns the position string at the top of the stack in the string at the second from the top of the stack.
SUBSTR	Pops two numbers of a string. LENGTH should be at top of the stack, START position at second from top of the string is at third from top. The substring of the string starting at START and LENGTH long is returned to the stack.
CATSTR	Concatenates the string at top of stack to the string at second to top of stack.
TRIMSTR	Trims all leading and trailing blanks from the string at top of stack.

Table 8-8. Effect of STRING Operators on the Stack

STRING	Stack Before Operation			Stack After Operation
	TOS[1]	TOS-1	TOS-2	TOS
LENGTH	string	-	-	length of string
NUM	“one”	-	-	111 (ASCII ’o’)
VAL	“1.2”	-	-	1.2 (a number)
VALSTR	1.2	-	-	“1.2”
CHRSTR	49	-	-	“1”
POS	little	big	-	POS(big,little)
SUBSTR	length	start	string	substr(of string)
CATSTR	str2	str1	-	str1str2
TRIMSTR	“ str ”	-	-	“str1”

[1] Top of Stack

BITS

The BITS statement is an action statement. This statement provides a way of extracting a bit pattern from a source. The HP ID pushes the resulting value on the stack.

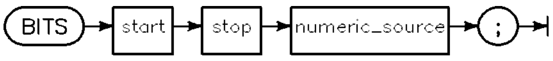
Multiple BITS statements may be used to accumulate a bit pattern from separate sources.

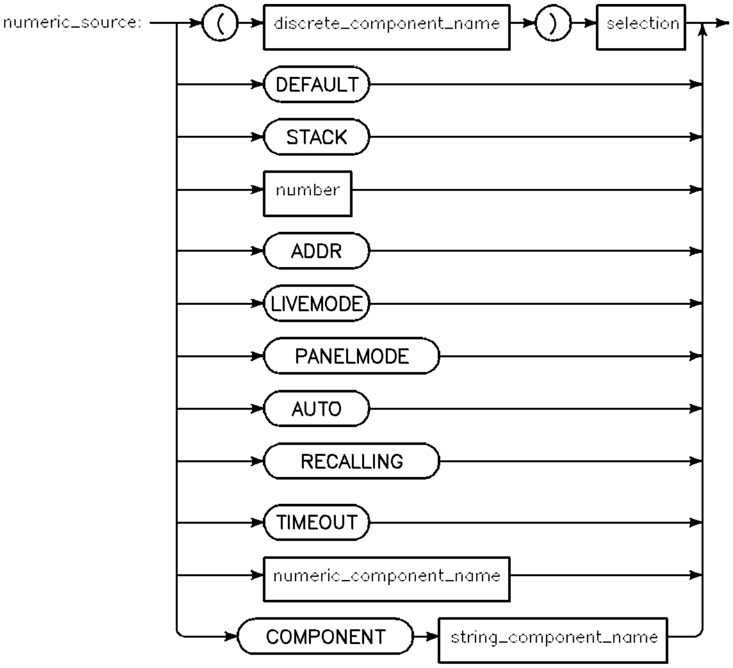
For any sequence of BITS instructions a single value is placed on the stack. That value is the accumulated value of the fields indicated with the list of BITS statements. Each BITS statement specifies a source and the desired number of bits from that source. The number of bits is equal to (stop-start+1) bits. These bits are taken from the source starting with the least significant bit of the source and put into the stack destination at position <start> through <stop>.

For example, `BITS 3,1,7` takes 3-1+1 (start-stop+1) from the source (7=0111 in binary) starting with the least significant bit. Thus, 111 is extracted from the bit pattern 0111 and is put into position 3 through 1 on the stack. Assuming a zero was already on the stack, this yields a 1110 on the stack (decimal 14). The bit field is specified as a start position and a stop position in the stack destination.

This bit field is extracted from the source and accumulated into the result, starting with the least significant bit and incrementing by the field width of each BITS statement. This statement validates all source components unless they are marked DONTCARE.

Syntax





Drawing 2

Item	Description
start	A numeric source (see drawing 2) that specifies the left bit of the pattern (0 origin from least significant bit, which is right-most bit).
stop	A numeric source (see drawing 2) that specifies the right bit of the pattern (0 origin from least significant bit, which is right-most bit).
numeric_source	See drawing 2. Specifies the source from which the bits are extracted.

BITS

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.

Example

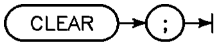
```
COMPONENT MASK;
TYPE INTEGER;
GET ACTIONS BUILD_MASK;
END COMPONENT;

ACTIONS BUILD_MASK;
  BITS 0,0 FAULT;
  ! Assume that we want to build this mask register:
  !
  !   BIT 0 (LSB)     FAULT
  !   BIT 1           POWER_ON
  !   BIT 2           FRONT_PANEL
  !   BITS 3-7        ERROR_MASK
  !
  BITS 1,1 POWER_ON;
  BITS 2,2 FRONT_PANEL;
  BITS 7,3 ERROR_MASK;
  STORE MASK;
END ACTIONS;
```

CLEAR

The CLEAR statement is an action statement. The CLEAR statement causes the development environment to send the HP-IB selected device clear command (SDC) to the instrument.

The CLEAR statement has no parameters.

Syntax



Example

```
SET ACTIONS;
   CLEAR; ! Clears the instrument
   POKEINITIAL ! Sets the components to INITIAL values
END ACTIONS;
```

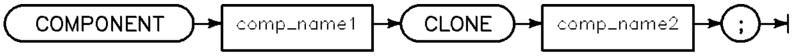
CLONE

CLONE is a clause that may follow the component name in a COMPONENT compound statement. It creates a component that differs from the cloned component only in name.

CLONE is useful when developing HP IDs for devices such as switches. It saves memory in the HP ID file by eliminating the need for you to completely specify all components that differ in name only.

The new component will have the same combination of NOTSAVED, NOGEN and NOERRCHECK as the original.

Syntax



Item	Description
comp_name1	The name of the new component.
comp_name2	The name of the component being cloned.

CLONE

In the following example, the instrument has four registers. They are named MAX, MIN, SDEV, AVG. The following four components access these registers.

Example

```
COMPONENT MAX;
  TYPE CONTINUOUS;
  INITIAL 0;
  SET ACTIONS;
    OUTPUT SELF FORMAT '"STORE",K';
  END ACTIONS;
  GET ACTIONS;
    OUTPUT SELF FORMAT '"RECALL",K';
    ENTER DEFAULT;
  END ACTIONS;
END COMPONENT;
!
COMPONENT MIN CLONE MAX;
COMPONENT SDEV CLONE MAX;
COMPONENT AVG CLONE MAX;
```

Cross Reference

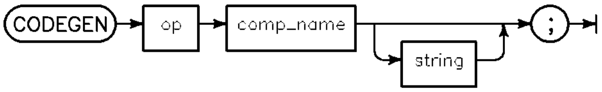
- COMPONENT

CODEGEN (HP ITG only)

The CODEGEN statement is an action statement. This statement will cause a specific line of code to be logged to the editor when it is executed in panels mode with Log HP ITG Calls mode enabled.

This statement can be used with NOGEN to override the default code generation algorithm used by HP ITG.

Syntax



Item	Description
op	Specifies which subprogram is called. If op = SET, HP ITG generates a call to hpt_set. If op = GET, HP ITG generates a call to hpt_get. If op = PEEK, HP ITG generates a call to hpt_peek. If op = POKE, HP ITG generates a call to hpt_poke. If op = PUSH, HP ITG generates a call to hpt_push. For SET, GET, POKE, and PEEK, the variation of the subprogram called depends on the TYPE of component. For example, for a STRING component with SET, HP ITG generates a call to hpt_set_str.
comp_name	The name of the component to be used in the subprogram call.
string	Optional. If specified, this string is used as the third parameter in the generated subprogram call rather than a value input by the user.

Example

```
REVISION 2.0;

COMPONENT LEARN_STRING NOTSAVED;
    TYPE STRING 128;
    INITIAL "";
    SET ACTIONS;
        OUTPUT LEARN_STRING FORMAT K;
        INVALIDATE ALL;
    END ACTIONS;
    GET ACTIONS;
        OUTPUT STRING "LP1";
        ENTER LEARN_STRING FORMAT K;
    END ACTIONS;
END COMPONENT;

PANEL MAIN;

    POSITION 1,1;
    SIZE 220, 120;

    TEXT 7,84, "Learn String";

    BUTTON LEARN_STRING;
        POSITION 120, 84;
        LABEL "Read";
        HIT ACTIONS;
            CODEGEN GET, LEARN_STRING;
            GET LEARN_STRING;
        END ACTIONS;
    END BUTTON;
```

(Continued)

```
    BUTTON LEARN_STRING;
        POSITION 165, 84;
        LABEL "Send";
        HIT ACTIONS;
            CODEGEN SET,LEARN_STRING,"Value";
            SET LEARN_STRING;
        END ACTIONS;
    END BUTTON;

END PANEL;
```

In this example, the instrument supports a binary learn mode feature. Because the string is binary and too long to display, it is useful to provide two buttons—one that reads the string from the instrument and a second that sends the string to the instrument. The default code generation would provide an hpt_push for each of these actions. We want it to generate an hpt_get_str and hpt_set_str. The CODEGEN statement in the example provides this behavior. Note that NOGEN was not necessary for the component LEARN_STRING because the CODEGEN statement is part of the HIT ACTIONS list, which suspends HP ITG's default behavior of performing SET ACTIONS.

Cross Reference

- NOGEN

COMPONENT

A COMPONENT ... END COMPONENT compound statement defines a component. As the name suggests, you describe the functionality of the component within this statement.

You must specify a name for the component and the type of value it will have so that the development environment can create an appropriate component. You can also specify a range or list of valid values, an initial value, the name of one or more components whose value depends on this component's value.

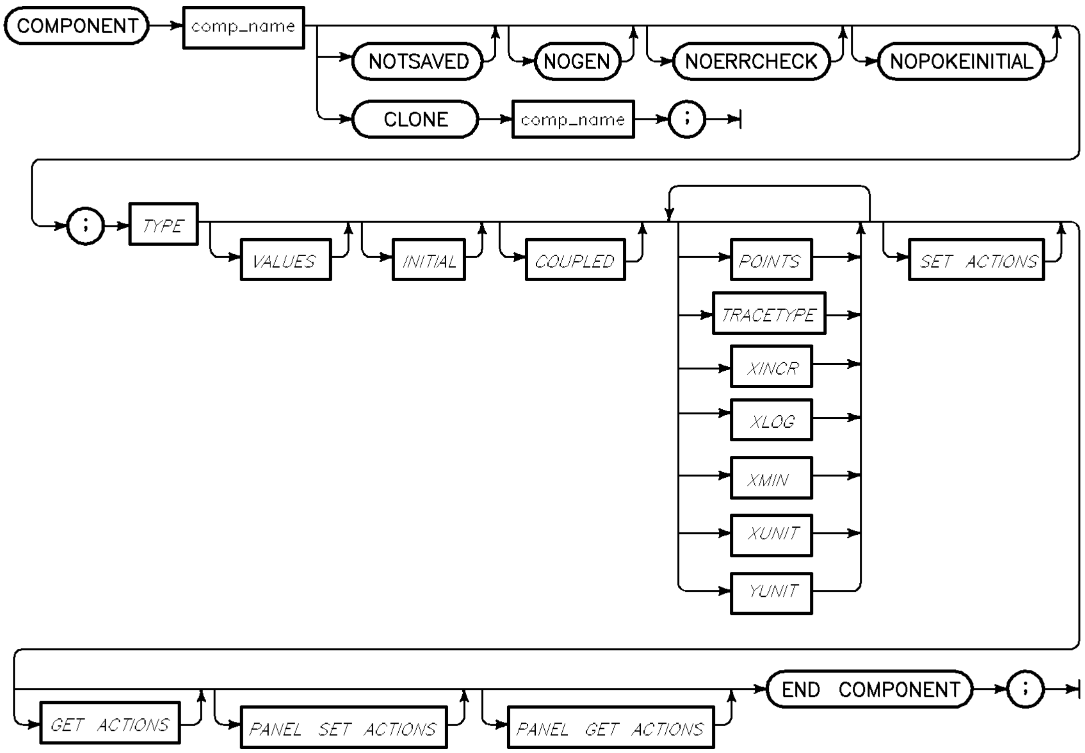
Component and action names must start with an alpha character, A-Z or a-z. That character may be followed by any mix of alphanumeric characters, either upper or lower case, or by an underscore. The syntax is not case sensitive.

Following is a list of possible component names:

- FREQUENCY
- SLOT4
- SLOT_4
- Start_Frequency

You can also add up to four action lists: one SET ACTIONS, one GET ACTIONS, one PANEL SET ACTIONS, and one PANEL GET ACTIONS.

Syntax


COMPONENT
comp_name
NOTSAVED
NOGEN
NOERRCHECK
NOPOKEINITIAL
CLONE
comp_name
;
;
TYPE
VALUES
INITIAL
COUPLED
POINTS
TRACETYPE
XINCR
XLOG
XMIN
XUNIT
YUNIT
SET ACTIONS
GET ACTIONS
PANEL SET ACTIONS
PANEL GET ACTIONS
END COMPONENT
;

8

Item	Description
comp_name	The name of the component. This is a required part of the component. Component and action names must start with an alpha character, A-Z or a-z. That character may be followed by any mix of alphanumeric characters, either upper or lower case, or by an underscore. The syntax is not case sensitive. Following is a list of possible component names: ■ FREQUENCY ■ SLOT4 ■ SLOT_4 ■ Start_Frequency
NOTSAVED	Optional. Instructs the development environment to not include the component when the user stores or recalls a state.
NOGEN	Optional. Instructs the development environment to not generate the default code associated with the component.
NOERRCHECK	Optional. Instructs the development environment to not perform default error checking after executing any actions initiated by the component.
CLONE	CLONE is a clause that may follow the component name in a COMPONENT compound statement. It creates a component that differs from the cloned component only in name.
TYPE	Required. Specifies the type of the component.
VALUES	Optional (except for DISCRETE components). Used to specify a list or range of valid values.
INITIAL	Optional. Used to specify the initial value of the component.
COUPLED	Optional. Used to specify one or more components that are linked to the current component.
SET ACTIONS	Optional. Used to specify the actions the development environment must execute whenever the component is referenced in an hpt_set call.

Item	Description
GET ACTIONS	Optional. Used to specify the actions the development environment must execute whenever the component is referenced in an hpt_get call.
PANEL SET ACTIONS	Optional. Used to specify the actions the development environment must execute after the component's SET ACTIONS are executed. Executed only in the development environment.
PANEL GET ACTIONS	Optional. Used to specify the actions the development environment must execute after the component's GET ACTIONS are executed. Executed only in the development environment.
TRACETYPE	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies the type of trace (spectrum, waveform, etc.)
POINTS	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies the current dynamic dimensions of the array.
XMIN	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies the X-axis value of the first data point.
XINC	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies the X-axis spacing between data points. If XLOG is true, specifies the number of steps per decade.
XLOG	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies whether the X-axis spacing is logarithmic.
XUNIT	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies the X-axis unit for graph labeling.
YUNIT	Optional. Valid for TYPE ITRACE and RTRACE only. Specifies the Y-axis unit for graph labeling.

Cross Reference

- CLONE
- COUPLED
- INITIAL
- NOERRCHECK
- NOGEN
- NOTSAVED
- TYPE
- VALUES

COUPLED

The COUPLED statement, which is used within the COMPONENT ... END COMPONENT compound statement, is designed to simplify handling the relationship between related components. The development environment allows you to couple a component with as many other components as necessary.

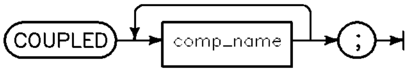
After the SET ACTIONS of the component containing the COUPLED statement are executed, the development environment acts upon the components specified in the COUPLED statement as follows:

- In the development environment, if Live mode is on, then the development environment executes the GET ACTIONS and PANEL GET ACTIONS of all components specified in the COUPLED statement. If Live mode is off, no action is taken.
- During run time, the development environment invalidates the components specified in the COUPLED statement so that the next recall sends all the components' values again.

When recalling a state, COUPLED has no effect.

In general, when a change in component A may change component B, you should include the statement COUPLED B in component A's declaration. You should not, however, couple to a component that has no GET ACTIONS.

Syntax



Item	Description
comp_name	The name of the component coupled to the current component.

Example

```
COMPONENT VOLT2;
  TYPE CONTINUOUS;
  VALUES RANGE 0,20,.2.0.006;
  INITIAL 0;
  COUPLED CURR2;
  SET ACTIONS;
    OUTPUT VOLT2 FORMAT ’"VSET 2,",DDD.DDDD’;
  END ACTIONS;
  GET ACTIONS;
    OUTPUT STRING "VSET? 2";
    ENTER VOLT2 FORMAT K;
  END ACTIONS;
END COMPONENT;
!
COMPONENT CURR2;
  TYPE CONTINUOUS;
  VALUES RANGE 0,10.3,0.05;
  INITIAL 0;
  COUPLED VOLT2;
  SET ACTIONS;
    OUTPUT CURR2 FORMAT ’"ISET 2,",DDD.DDDD’;
  END ACTIONS;
  GET ACTIONS;
    OUTPUT STRING "ISET? 2";
    ENTER CURR2 FORMAT K;
  END ACTIONS;
END COMPONENT;
```

This example shows that when the voltage is changed the current may change as a result.

In the development environment with Live mode on, the development environment will get the new current after performing the SET ACTIONS of VOLT2. This provides the user with good feedback as to the state of the instrument. During run time, the development environment just invalidates the component.

Cross Reference

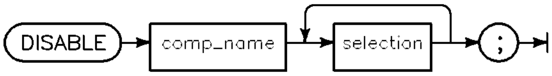
- COMPONENT

DISABLE

The DISABLE statement is an action statement. It is used to disable specified selections of a TYPE DISCRETE component. In the development environment, the user can enter only nondisabled selections (that is, they are enabled unless you disable them). Once disabled, you must use the ENABLE statement to reenable them.

Note that POKEINITIAL does not reenable components, so the component specified by INITIALIZE COMPONENT should reenable any necessary selections.

Syntax



Item	Description
comp_name	The name of the DISCRETE component whose list of values you want to alter.
selection	One or more selections of a DISCRETE component that you want to disable.

Example

```
REVISION 2.0;

COMPONENT MENU NOTSAVED NOGEN NOERRCHECK;
  TYPE DISCRETE;
  VALUES MAIN,STATUS,RELAY;
  !.
  !.
  !.
END COMPONENT;

COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
    !.
    !.
    OUTPUT STRING "RELAY?";
    ENTER STACK FORMAT K;
    SELECT STACK;
      CASE 0;
        HIDE RELAY_P;
        DISABLE MENU,RELAY;
      CASE 1;
        SHOW RELAY_P;
        ENABLE MENU,RELAY;
    END SELECT;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- ENABLE

DONTCARE

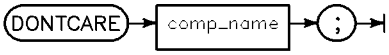
The DONTCARE statement is an action statement. This statement is one of three that control the status of a component. The other two are the INVALIDATE and VALIDATE statements.

The development environment's incremental state programming feature enables it to send only the commands needed to configure the instrument as specified in the recalled state. To do this, the development environment keeps track of the current values of all components and so, can compare the current value with the specified value of any component in the state.

There may be some components that you don't want the development environment to send when recalling a state; normally these components should be specified as NOTSAVED. Occasionally, you will have a component that is necessary at some times but not at others. To do this, you can set the status of a component to DONTCARE. When a state is recalled that includes a component marked as DONTCARE, the development environment will not execute the SET actions for that component.

A component ceases to be DONTCARE when it is validated (with VALIDATE), invalidated (with INVALIDATE), or when development environment recalls a state that includes the component and it is not marked as DONTCARE in the state being recalled.

Syntax



Item	Description
comp_name	The name of the component whose value you don't want the development environment to compare to the value specified in the recalled state.

Example

```
REVISION 2.0;

COMPONENT ARANGE;
    TYPE DISCRETE;
    VALUES OFF,ON;
    INITIAL ON;
    SET ACTIONS;
        SELECT ARANGE;
        CASE OFF;
            OUTPUT STRING "AERH";
            INVALIDATE RANGE;
        CASE ON;
            OUTPUT STRING "AERA";
            DONTCARE RANGE;
        END SELECT;
    END ACTIONS;
END COMPONENT;

COMPONENT RANGE;
    TYPE INTEGER;
    VALUES RANGE 1,5;
    INITIAL DONTCARE;
    SET ACTIONS;
        OUTPUT RANGE FORMAT '"AERM",D,"EN"';
        FETCH (ARANGE)OFF;
        STORE ARANGE;
        VALIDATE RANGE;
    END ACTIONS;
END COMPONENT;
```

(Continued)

```
PANEL MAIN;
  POSITION 1,1;
  SIZE 200,200;
  DISCRETE ARANGE;
    POSITION 90,20;
    TITLE "A Range";
  END DISCRETE;
  CONTINUOUS RANGE;
    POSITION 90,50;
    TITLE "Range";
  END CONTINUOUS;
END PANEL;
```

Cross Reference

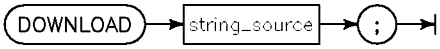
- INVALIDATE
- NOTSAVED
- VALIDATE

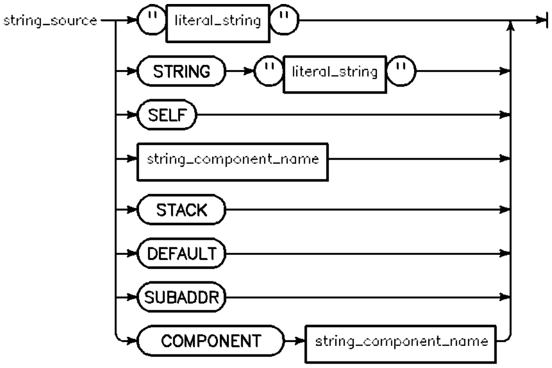
DOWNLOAD

This statement causes the data contained in a file on the host computer to be transferred to the instrument. The file is transferred byte for byte and the development environment performs no data processing function.

The file name is specified in the string source, and is interpreted by the file system of the host computer.

Syntax





Item	Description
string_source	The name of the file to be downloaded to the instrument.

Example

```
COMPONENT SEND_HOP_LIST NOTSAVED;
  TYPE STRING 32;
  SET ACTIONS;
    FETCH SEND_HOP_LIST;
    FETCH "";
    NE;
    IF STACK THEN;
      OUTPUT STRING "HOP:LIST?"
      DOWNLOAD SEND_HOP_LIST;
    END IF;
  END ACTIONS;
END COMPONENT;

...

PANEL HOP;
  ...
  INPUT SEND_HOP_LIST;
    POSITION 4,55;
    SIZE 120,19;
    STYLE "FILESELECT";
  END INPUT;
```

Cross Reference

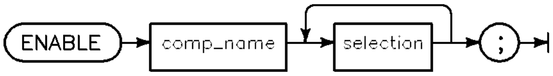
- UPLOAD

ENABLE

The ENABLE statement is an action statement. It is used to ENABLE specified selections of a TYPE DISCRETE component. In the development environment, the user can enter only enabled selections (that is, they are enabled unless you disable them). Once disabled, you must use the ENABLE statement to reenable them. Initially, all selections for a DISCRETE are ENABLED.

Note that POKEINITIAL does not reenable components, so the component specified by INITIALIZE COMPONENT should reenable any necessary selections.

Syntax



Item	Description
comp_name	The name of the DISCRETE component whose list of values you want to alter.
selection	One or more selections of a DISCRETE component that you want to enable.

Example

```
REVISION 2.0;

COMPONENT MENU NOTSAVED NOGEN NOERRCHECK;
  TYPE DISCRETE;
  VALUES MAIN,STATUS,RELAY;
  !.
  !.
  !.
END COMPONENT;

COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
    !.
    !.
    OUTPUT STRING "RELAY?";
    ENTER STACK FORMAT K;
    SELECT STACK;
      CASE 0;
        HIDE RELAY_P;
        DISABLE MENU,RELAY;
      CASE 1;
        SHOW RELAY_P;
        ENABLE MENU,RELAY;
    END SELECT;
  END ACTIONS;
END COMPONENT;
```

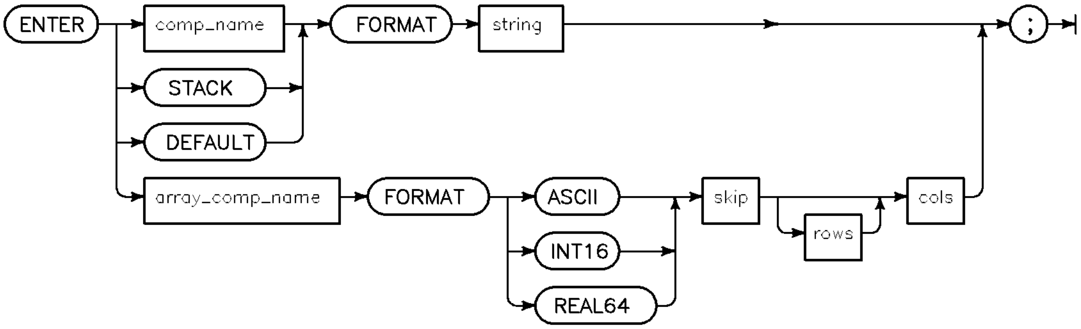
Cross Reference

- DISABLE

ENTER

The ENTER statement is used to input data from the instrument. You can specify that the development environment put the data into any component TYPE.

Syntax



Item	Description
comp_name	Name of an INTEGER, DISCRETE, STRING or CONTINUOUS component into which the development environment puts the incoming data.
STACK	The development environment puts the incoming data at the top of the stack. The data must be numeric, not string.
DEFAULT	The development environment replaces the value of the current component with the incoming data.
string	See ACTIONS keyword table 8-7 for specifiers.
array_comp_name	Name of an ITRACE, RTRACE, IARRAY, or RARRAY component into which the development environment puts the incoming data. The development environment converts the data to the proper type.
ASCII	Specifies that the development environment will read items separated by commas or carriage return/line feed.
INT16	Specifies that the development environment will read 16-bit integer data.
REAL64	Specifies that the development environment will read 64-bit IEEE 754 floating point data.
skip	A constant or name of a component whose value specifies how many bytes to skip before putting the data into `array_comp_name`.
rows	A constant or name of a component whose value specifies how many elements (that is, not bytes) to fill up with the incoming data. If **rows** is missing, it defaults to 1.
cols	A constant or name of a component whose value specifies how many elements (that is, not bytes) to fill up with the incoming data.

For array enters, **rows** and **cols** indicate whether they are entered with the row or column index incrementing faster. If rows and cols match the rows and cols specified in the IARRAY or RARRAY component statement, then the array is entered with the column index incrementing faster. If the rows and cols in the ENTER statement are swapped from those specified in the IARRAY or RARRAY component statement, then the array is entered with the row index incrementing faster. If rows or cols in the ENTER statement is smaller than rows or cols in the array component, then only that many rows or columns is ENTERED.

In any case, the smaller of rows or cols in the ENTER statement must match rows in the TYPE statement.

Table 8-9. Valid ENTER FORMAT Specifiers

FORMAT Specifier	Meaning
K	Freefield entry **Numeric:** Entered characters are sent to the number builder. Leading non-numeric characters are ignored. All blanks are ignored. Trailing non-numeric characters and characters sent with `EOI true` are delimiters. Numeric characters include digits, decimal point, +, -, e, and E when their order is meaningful. **String:** Entered characters are placed in the string. Carriage return not immediately followed by line feed is entered into the string. Entry to a string terminates on CR/LF, LF, a character received with `EOI true`, or when the dimensioned length of the string is reached.
D	Enters a character. Non-numerics are accepted to fill the character count. Blanks are ignored. Other non-numerics are delimiters.
Z	Same as D.
M	Same as D.
S	Same as D.
.	Same as D.
E	Same as DDDD.
ESZ	Same as DDD.
ESZZ	Same as DDDD.
ESZZZ	Same as DDDDD.
A	Enters a string character. Any character received is placed in the string.
X	Skips a character.
#	Statement is terminated when the last ENTER item is terminated. EOI and line feed are item terminators, and early termination is not allowed.
B	Demands one byte. The byte becomes a numeric quantity.
W	Demands one 16-bit word, which is interpreted as a 16-bit, two's complement integer.

Example

```
REVISION 2.0;

COMPONENT SINGLE;
TYPE INTEGER;
GET ACTIONS;
ENTER SINGLE FORMAT "K";
END ACTIONS;
END COMPONENT;

COMPONENT ARRAY;
TYPE RARRAY 1 20;
GET ACTIONS;
ENTER ARRAY FORMAT ASCII 0 1 20; ! SKIP
END ACTIONS;
END COMPONENT;
```

EOL

The EOL statement is a general statement that allows you to select the end-of-line (EOL) characters that are sent to the instrument.

The default EOL sequence is 13,10 (that is, carriage return/line feed).

You can specify up to two numbers, each of which is the decimal value of the desired ASCII character (see table 8-10).

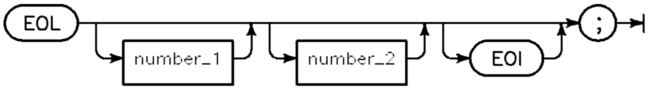
You can also specify EOI (end-or-identify) as part of the EOL sequence. This causes the HP-IB EOI line to be asserted with the last byte of the transmission. If EOL EOI is specified, then the EOI line is asserted with the last data byte.

The development environment buffers outgoing data before it is sent. If the development environment determines that it is time to send the buffer, and if the buffer is not empty, it is sent with the EOL sequence on the end.

The following action statements and an END ACTIONS cause the buffer to be sent:

- SET
- GET
- WAIT TIME
- WAIT SPOLL BIT
- ENTER
- TRIGGER
- SPOLL
- USERSUB
- CLEAR
- FLUSH
- POKEINITIAL

Syntax



EOL

Item	Description	Range
number_1	Optional. An integer. This is the decimal equivalent to the ASCII character that is the first character in the EOL sequence.	0 - 127
number_2	Optional. An integer. This is the decimal equivalent to the ASCII character that is the second character in the EOL sequence.	0 - 127
EOI	Optional. When specified, the development environment asserts EOI with the last data byte.	

Table 8-10. Decimal-ASCII Conversions

Control		Number & Symbol		Upper Case Letters & Symbols		Lower Case Letters & Symbols	
Dec.	**ASCII**	**Dec.**	**ASCII**	**Dec.**	**ASCII**	**Dec.**	**ASCII**
0	NUL	33	!	65	A	97	a
1	SOH	34	”	66	B	98	b
2	STX	35	#	67	C	99	c
3	ETX	36	$	68	D	100	d
4	EOT	37	%	69	E	101	e
5	ENQ	38	&	70	F	102	f
6	ACK	39	’	71	G	103	g
7	BEL	40	(	72	H	104	h
8	BS	41	)	73	I	105	i
9	HT	42	*	74	J	106	j
10	LF	43	+	75	K	107	k
11	VT	44	,	76	L	108	l
12	FF	45	-	77	M	109	m
13	CR	46	.	78	N	110	n
14	SO	47	/	79	O	111	o
15	SI	48	0	80	P	112	p
16	DL	49	1	81	Q	113	q
17	DCI	50	2	82	R	114	r
18	DC2	51	3	83	S	115	s
19	DC3	52	4	84	T	116	t
20	DC4	53	5	85	U	117	u
21	NAK	54	6	86	V	118	v
22	SYN	55	7	87	W	119	w
23	ETB	56	8	88	X	120	x
24	CAN	57	9	89	Y	121	y
25	EM	58	:	90	Z	122	z
26	SUB	59	;	91	[	123	{
27	ESC	60	<	92	\	124	|
28	FS	61	=	93	]	125	}
29	GS	62	>	94	ˆ	126	˜
30	RS	63	?	95	_	127	DEL\(rubout)
31	US	64	@	96	‘		
32	SP						

Example

```
! Example 1:  End with EOI
REVISION 2.0;
EOL EOI;
!
! Example 2:  End with a line feed
REVISION 2.0;
EOL 10;
```

Cross Reference

- SKIP EOL

ERROR COMPONENT

The ERROR COMPONENT statement allows you to take advantage of an instrument's error-checking capability. This statement indicates to the development product how to check for instrument errors. If the error checking is turned on, error checking is done after each transaction between the development product and the instrument.

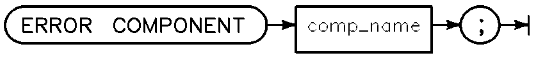
The development environment checks for instrument errors by executing the GET actions of the specified component. If the component is non-zero after executing the GET actions, an error is raised.

The GET ACTIONS list of the component indicated in the ERROR COMPONENT statement typically includes an OUTPUT statement

that queries the instrument's error register, and an ENTER statement in which the instrument returns a value indicating whether an error has occurred.

When error checking is enabled during run time and an error is detected, an error will be generated. (In HP ITG only, the routine hpt_errmsg can then be used to ascertain more specifics about the error.)

Syntax



Item	Description
comp_name	The component whose GET ACTIONS are executed after each transaction with the instrument. This must be a TYPE INTEGER or CONTINUOUS component.

Example

```
REVISION 2.0;
!
ERROR COMPONENT SYSTEM_ERROR;
!     .
!     .
!     .
COMPONENT SYSTEM_ERROR NOTSAVED;
  TYPE INTEGER;
  GET ACTIONS;
    OUTPUT STRING "ERR?";
    ENTER SYSTEM_ERROR FORMAT K;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

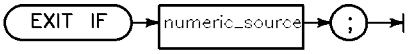
- NOERRCHECK

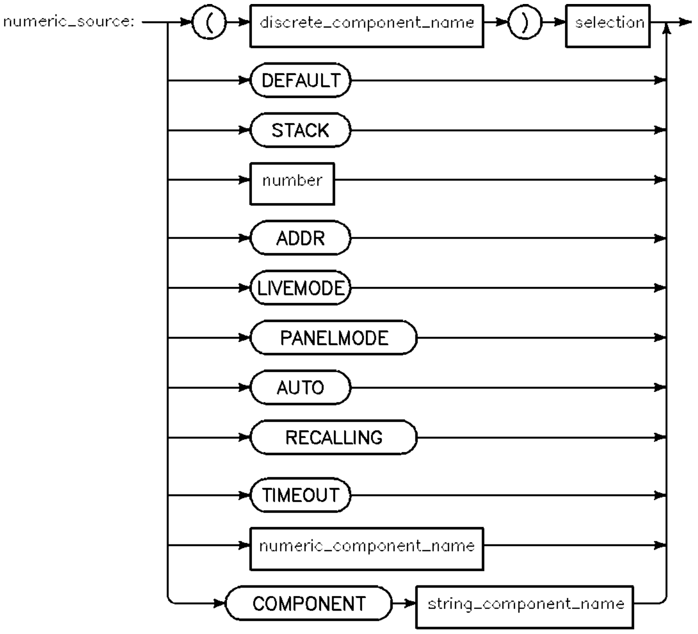
EXIT IF

The EXIT IF statement is an action statement. It is used to break out of a LOOP ... END LOOP compound statement. EXIT IFs can occur anywhere inside a loop.

The EXIT IF condition will cause control to be transferred to the statement after the END LOOP statement of the LOOP where the EXIT IF occurs.

Syntax





Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.

Cross Reference

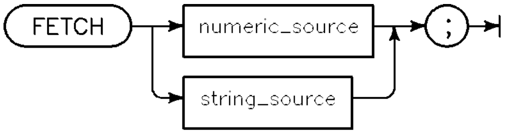
- LOOP

FETCH

The FETCH statement is an action statement. This statement, along with the STORE statement, and arithmetic, logic, and string operators form the vocabulary of a FORTH-like stack machine.

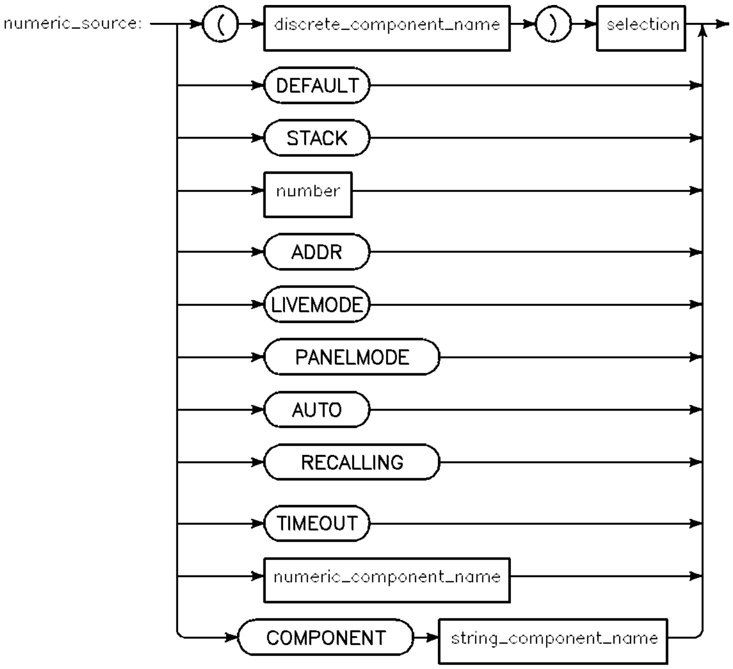
The FETCH statement places the value from the source on top of the stack (that is, it pushes the value from the source).

Syntax



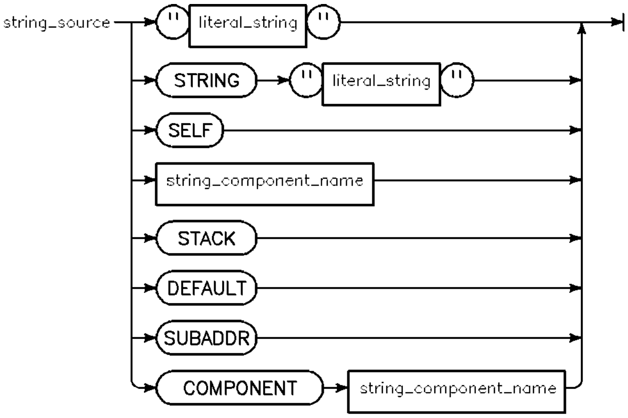
Item	Description
numeric_source	See drawing 2.
string_source	See drawing 3.

FETCH



Drawing 2

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.



Drawing 3

In either case, the array element is FETCHed onto the stack.

Item	Description
STRING	The development environment uses a user-entered string as the value.
SELF	The development environment uses the name of the current component as the value.
string_component_name	The development environment uses the name of the STRING component specified.
STACK	The development environment uses the value at the top of the stack.
DEFAULT	The development environment uses the value of the current component.
SUBADDR	The development environment uses the subaddress the user typed into the configuration box.
COMPONENT string	Indicates using a component as the source component name.

Fetching Array Elements

Array elements can be FETCHed onto the stack. Arrays are either one-dimension or two-dimension, and start at one. To FETCH one element of an array, use the following statements in an action list:

- **One-dimension array:**
  - □ FETCH index
  - □ FETCH comp_name
- **Two-dimension array:**
  - □ FETCH row_index
  - □ FETCH col_index
  - □ FETCH comp_name

Example

```
FETCH SRQ_MASK; ! Puts value on stack
FETCH 0; ! Puts value on stack
BIT; ! Puts the 0-position bit of SRQ_MASK on stack
STORE SRQ_FAULT; ! and stores it in SRQ_FAULT
```

FLUSH

The FLUSH statement is an action statement. FLUSH is used to force the development environment to output its buffer.

The development environment buffers outgoing data before it is sent. The FLUSH statement is used to force the buffer to be transmitted to the instrument. If the buffer is not empty, the EOL sequence will be sent also.

The following ACTIONS statements and END ACTIONS cause the development environment to flush the buffer:

- SET
- GET
- WAIT TIME
- WAIT SPOLL BIT
- ENTER
- TRIGGER
- SPOLL
- USERSUB
- CLEAR
- FLUSH
- POKEINITIAL

FLUSH does not have any parameters.

Example

In the following example, let us assume that the default EOL sequence is being used.

```
SET ACTIONS;
  OUTPUT "FUNC ";
  OUTPUT FUNCTION TABLE "DCV","ACV","OHM";
END ACTIONS;
```

Because there is not a FLUSH statement between the two OUTPUT statements, `FUNC DCV` C_R L_F is sent to the instrument. If a FLUSH statement is added between the two output statements, `FUNC` C_R L_F `DCV` C_R L_F would be sent.

Note that adding a second FLUSH between the OUTPUTs would not have sent the EOL sequence again. This is because the buffer is empty at this point.

It is desirable to use the FLUSH statement when sending two separate commands as in the following example:

```
GET ACTIONS;
  OUTPUT "INIT OFF";
  FLUSH;
  OUTPUT "FUNC?";
  ENTER FUNCTION;
END ACTIONS;
```

Note that a FLUSH statement is not required before the ENTER statement.

Cross Reference

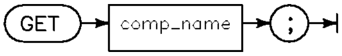
- ENTER
- EOL
- OUTPUT
- CLEAR
- SPOLL
- TRIGGER

GET

The GET statement is an action statement. When the GET statement is executed in an action list, the development environment executes the GET ACTIONS of the component specified in the statement. When the development environment is running, it then executes the PANEL GET ACTIONS, if there are any.

You can nest GET statements up to the available memory on your computer.

Syntax



Item	Description
comp_name	The name of the component whose GET ACTIONS are executed when the GET statement is encountered in an action list.

Example

```
ACTIONS FINISH_POKEINITIAL;
! Assume that a POKEINITIAL was just executed.
! This gets the values for all components not initialized.
  GET SENS_VOLT1;
  GET SENS_CURR1;
  GET CURR1;
!    .
!    .
!    .
END ACTIONS;
```

Cross Reference

- ACTIONS

GET ACTIONS

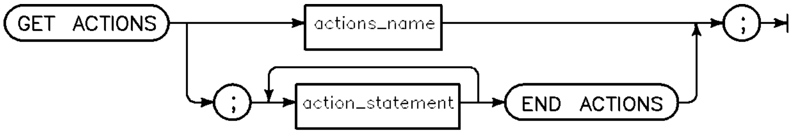
The GET ACTIONS ... END ACTIONS compound statement defines an action list that will GET the value of a component.

The GET ACTIONS statement is part of a component description and is typically used to instruct the instrument to make a measurement or to query the instrument for the value of a component.

The GET ACTIONS are executed when:

- hpt_get, hpt_get_str, hpt_get_ary, or hpt_get_iary is executed on this component.
- hpt_get2, hpt_get_str2, hpt_get_ary2, or hpt_get_iary2 is executed on this component.
- A GET statement references this component in an action list.
- A component that is coupled to this component is changed in the development environment with Live mode on.

Syntax



Item	Description
actions_name	The name of the action list.
action_statement	See tables 8-1 through 8-8 under the ACTIONS keyword for the valid statements.

Example

```
ACTIONS READING;
  SELECT trigger;
  CASE SINGLE;
     OUTPUT STRING "TR2;
  CASE FAST;
     OUTPUT STRING "TR1";
  CASE ELSE;
  END SELECT;
  ENTER READING FORMAT "K";
END ACTIONS;
!
COMPONENT SAMPLE;
  TYPE CONTINUOUS;
  GET ACTIONS READING;
END COMPONENT;
```

Cross Reference

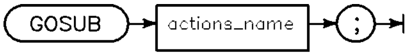
- ACTIONS

GOSUB

The GOSUB statement is an action statement. GOSUB statement is used to call a named action list.

You can nest GOSUBs up to the memory limitations of your computer.

Syntax



Item	Description
actions_name	The name of the named action list.

Example

```
REVISION 2.0;

ACTIONS FREQ_0;
   FETCH 0;
   STORE FREQ;
END ACTIONS;

COMPONENT DC;
  TYPE INTEGER;
  SET ACTIONS;
    OUTPUT STRING "DC";
    GOSUB FREQ_0;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

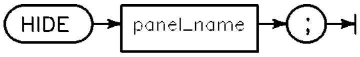
- ACTIONS

HIDE

The HIDE statement is an action statement. This statement allows you to specify that a subpanel is not displayed. It is executed only in the development environment.

HIDE is hierarchical. If you HIDE panel_name, then all subpanels of panel_name will not be seen, but when you SHOW panel_name they will reappear.

Syntax



Item	Description
panel_name	The name of the subpanel that you do not want displayed.

In the following example, HP_438 is the main panel. TOP and WATT_READING are its subpanels. SUB_WATT is a subpanel of WATT_READING. When WATT_READING is hidden, SUB_WATT will also disappear.

Example

```
INITIALIZE COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
!    .
!    .
!    .
  END ACTIONS;
  PANEL SET ACTIONS;
    SHOW TOP;
    HIDE WATT_READING;
    INVALIDATE A_RANGE;
  END ACTIONS;
END COMPONENT;
!     .
!     .
!     .
!Panel section
PANEL HP_438;
  PANEL TOP;
    POSITION 0,182;
!     .
!     .
!     .
  END PANEL;
  PANEL WATT_READING;
    POSITION 10,150;
    PANEL SUB_WATT;
!     .
!     .
!     .
    END PANEL;
  END PANEL;
!     .
!     .
!     .
END PANEL;
```

Cross Reference

- SHOW

IF ... END IF

The IF ... END IF compound statement is an action statement. The IF ... END IF statement allows you to conditionally execute a portion of your action lists.

You can nest IF ... END IF, LOOP ... END LOOP, and SELECT ... END SELECT compound statements up to ten deep.

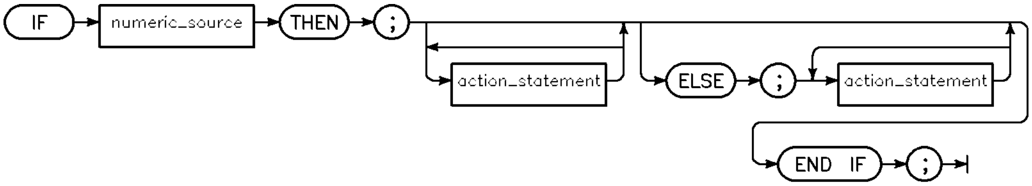
If the value is not zero:

- Then the development environment executes the statements between THEN and ELSE (or END IF, if ELSE is not present).

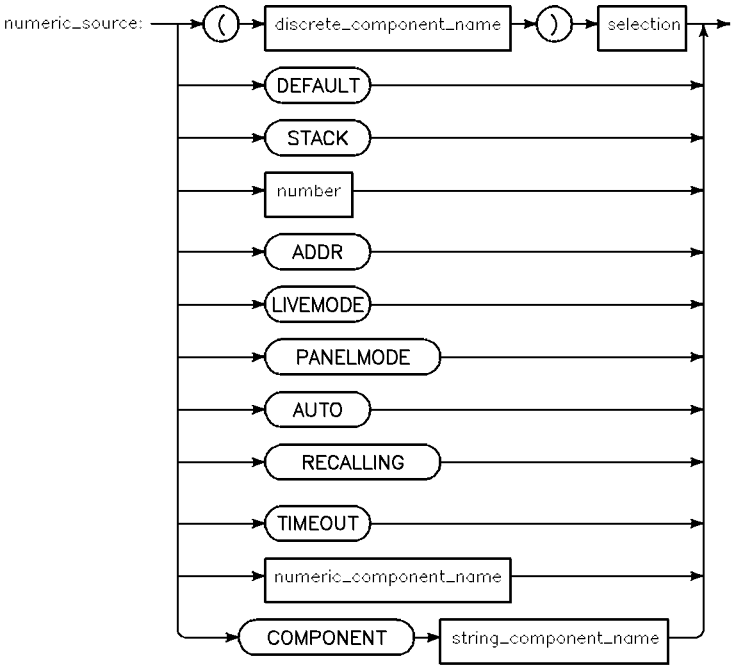
If the value is zero:

- Then the development environment executes the statements between ELSE and END IF if ELSE is present.

Syntax



Item	Description
numeric_source	See drawing 2. The source provides a value.
action_statement	See tables 8-1 through 8-8 under the ACTIONS keyword for the valid statements.


numeric_source:
(
discrete_component_name
)
selection
DEFAULT
STACK
number
ADDR
LIVEMODE
PANELMODE
AUTO
RECALLING
TIMEOUT
numeric_component_name
COMPONENT
string_component_name

Drawing 2

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.

Example

```
REVISION 2.0;

COMPONENT MENU NOTSAVED NOGEN NOERRCHECK;
  TYPE DISCRETE;
  VALUES MAIN,STATUS,RELAY;
  !.
  !.
  !.
END COMPONENT;

COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
    !.
    !.
    OUTPUT STRING "RELAY?";
    ENTER STACK FORMAT K;
    IF STACK THEN;
        HIDE RELAY_P;
        DISABLE MENU,RELAY;
    ELSE;
        SHOW RELAY_P;
        ENABLE MENU,RELAY;
    END IF;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- SELECT

INITIAL

The INITIAL statement allows you to specify an initial value for the component as well as an initial status. It is an optional part of a component description, and if present, must follow the VALUES statement.

The development environment sets each component to the values specified in its INITIAL statement when executing a POKEINITIAL statement.

For status, you can specify the component be initially INVALID or DONTCARE. The default is VALID. If you specify a state, you may also specify a value.

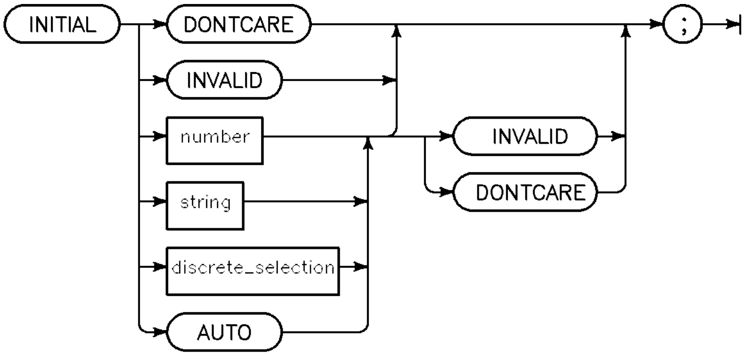
The development environment ignores INITIAL values set for IARRAY and RARRAY components. When a program is run, the development environment sets IARRAY and RARRAY values to 0. The POKEINITIAL statement ignores IARRAY and RARRAY components.

Defaults:

- DISCRETE components are set to the first value in their VALUES list.
- STRING components are set to " " (empty string).

All others are set to 0.

Syntax



Item	Description
number	An integer or real number that you want to be the initial value of an INTEGER or CONTINUOUS component.
string	A string that you want to be the initial value of a STRING component.
discrete_selection	A value from the list of valid values specified in the VALUES statement of a DISCRETE component.
DONTCARE	Specifies that the component is initially DONTCARE.
INVALID	Specifies that the component is initially INVALID.
AUTO	Specifies that the component, which must be CONTINUOUS and include AUTO in its VALUE statement, initially has the value AUTO.

Example

```
REVISION 2.0;
INITIALIZE COMPONENT RESET; ! Executes SET ACTIONS of component RESET.
!     .
!     .
!     .
COMPONENT UNITS;
  TYPE DISCRETE;
  VALUES LOG,LINEAR;
  INITIAL LOG;  ! Sets UNITS to LOG when POKEINITIAL executed.
!     .
!     .
!     .
END COMPONENT;
!
```

(Continued)

INITIAL

```
COMPONENT A_CAL_ADJUST;
  TYPE CONTINUOUS;
  VALUES RANGE 50,120,.1;
  INITIAL 100 ! Sets A_CAL_ADJUST to 100 when POKEINITIAL executed.
END COMPONENT;
!
COMPONENT A_CAL NOTSAVED;
  TYPE INTEGER; ! Defaults to initial value of 0.
  SET ACTIONS;
    OUTPUT A_CAL_ADJUST FORMAT '"AECL",K,"ENEN"';
  END ACTIONS;
END COMPONENT;
!
COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
    CLEAR;
!     .
!     .
!     .
    POKEINITIAL;  !Sets components to initial value.
  END ACTIONS;
END COMPONENT;
```

Cross Reference

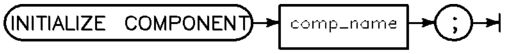
- COMPONENT
- POKEINITIAL
- VALUES

INITIALIZE COMPONENT

The INITIALIZE statement indicates to the development environment which component should be SET to initialize the HP ID and instrument. The development environment sets the INITIALIZE component whenever the user adds the HP ID to a test system in the development environment or executes hpt_assign in the run-time environment.

Usually, the INITIALIZE component will include the POKEINITIAL statement.

Syntax



Item	Description
comp_name	This is the name of the INTEGER component whose SET ACTIONS the development environment executes whenever a program is run or whenever the HP ID is added to the development environment.

Example

See INITIAL.

Cross Reference

- POKEINITIAL
- NOPOKEINITIAL

INVALIDATE

The INVALIDATE statement is an action statement. This statement is one of three that control the state of a component. The other two are the DONTCARE and VALIDATE statements.

INVALIDATE is used to indicate to the development environment that the value of a component does not necessarily reflect the configuration of the instrument. This way, when the development environment recalls a state, it will know that this component needs to be SET again.

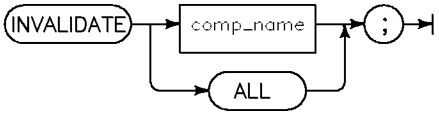
Using the INVALIDATE statement allows you to indicate that a setting in the instrument has changed without incurring the overhead of sending it again immediately (note that in many cases recall would have had to send this component again anyway).

The development environment's incremental state programming feature uses INVALIDATE also (see "How Recall Works" in chapter 6, "Advanced Topics").

You can use INVALIDATE in a SET ACTION to cause a companion component to be output during a state recall.

You can use INVALIDATE to indicate that a component is not longer DONTCARE, but the value in the component does not necessarily match that of the instrument.

Syntax



Item	Description
comp_name	The name of the component you want to mark as INVALID.
ALL	You can use this keyword to mark all components in the HP ID as INVALID.

Example

```
REVISION 2.0;

COMPONENT ARANGE;
    TYPE DISCRETE;
    VALUES OFF,ON;
    INITIAL ON;
    SET ACTIONS;
        SELECT ARANGE;
        CASE OFF;
            OUTPUT STRING "AERH";
            INVALIDATE RANGE;
        CASE ON;
            OUTPUT STRING "AERA";
            DONTCARE RANGE;
        END SELECT;
     END ACTIONS;
END COMPONENT;

COMPONENT RANGE;
    TYPE INTEGER;
    VALUES RANGE 1,5;
    INITIAL DONTCARE;
    SET ACTIONS;
        OUTPUT RANGE FORMAT ’"AERM",D,"EN"’;
        FETCH (ARANGE)OFF;
        STORE ARANGE;
        VALIDATE RANGE;
    END ACTIONS;
END COMPONENT;
```

(Continued)

```
PANEL MAIN;
  POSITION 1,1;
  SIZE 200,200;
  DISCRETE ARANGE;
    POSITION 90,20;
    TITLE "A Range";
  END DISCRETE;
  CONTINUOUS RANGE;
    POSITION 90,50;
    TITLE "Range";
  END CONTINUOUS;
END PANEL;
```

Cross Reference

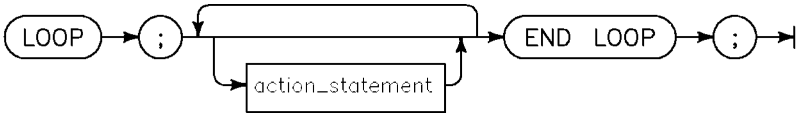
- DONTCARE
- VALIDATE

LOOP . . . END LOOP

The LOOP and END LOOP statements are action statements. These statements define a loop structure that can be exited with one or more EXIT IF statements.

Loops can be nested and the EXIT IF can occur anywhere within the LOOP. The EXIT IF condition will cause control to be transferred to the statement after the END LOOP statement of the loop where the EXIT IF occurs. You can next LOOP . . . END LOOP, IF . . . END IF, and SELECT . . . END SELECT compound statements up to 10 deep.

Syntax



Example

```
FETCH 10;
LOOP;
  FETCH 1;
  SUB;
  DUP;
  FETCH 0;
  EQ;
  EXIT IF STACK;
.
.
.
END LOOP
DROP;
```

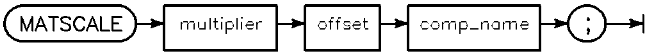
Cross Reference

- EXIT IF

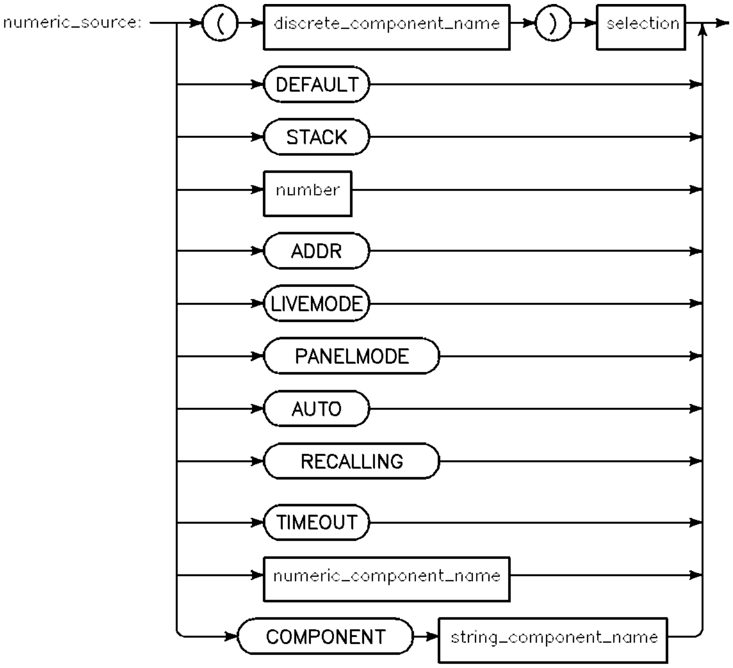
MATSCALE

The MATSCALE statement is an action statement that operates on TYPE ARRAY or RARRAY components. The MATSCALE statement is used to scale an array. It instructs the development environment to perform mx + b arithmetic on the specified component.

Syntax



Item	Description
multiplier	A numeric source that specifies the m value. You can use any of the values in drawing 2 (following page).
offset	A numeric source that specifies the b value. You can use any of the values in drawing 2 (following page).
comp_name	The name of an IARRAY or RARRAY component whose values will be scaled by **multiplier** and **offset**.


numeric_source:
(
discrete_component_name
)
selection
DEFAULT
STACK
number
ADDR
LIVEMODE
PANELMODE
AUTO
RECALLING
TIMEOUT
numeric_component_name
COMPONENT
string_component_name

Drawing 2

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.

Example

```
REVISION 2.0;
!
COMPONENT WF_CH1 NOTSAVED;
     TYPE RARRAY 1024;
     INITIAL INVALID;
     GET ACTIONS;
          OUTPUT STRING "WF_SIZE?";
          ENTER WF_SIZE FORMAT "K";
          OUTPUT STRING "YORIGIN?";
          ENTER YORIGIN FORMAT "K";
          OUTPUT STRING "YINCREMENT?";
          ENTER YINCREMENT FORMAT "K";
          OUTPUT STRING ":WAV:DATA?;";
          ENTER WF_CH1 FORMAT INT16 0 WF_SIZE;
          MATSCALE YINCREMENT,YORIGIN WF_CH1; ! Scales y-data in array
     END ACTIONS;
END COMPONENT;
```

NOERRCHECK

NOERRCHECK is an optional clause that follows the component name in the component description. The NOERRCHECK option instructs the development environment to not execute the GET ACTIONS list of the error component after transactions between the HP ID and the instrument that this component initiated.

This is useful when the error check would somehow interfere with data already in the instrument output buffer.

Cross Reference

- COMPONENT
- ERROR COMPONENT
- NOGEN
- NOPOKEINITIAL
- NOTSAVED

NOGEN (HP ITG only)

NOGEN is an optional clause that follows the component name in the component description. The NOGEN option instructs HP ITG to not generate any code associated with the component when the user is in the development environment.

You should specify the NOGEN option for components that create menus so that going from subpanel to subpanel does not generate code. These components should also be specified as NOTSAVED and NOERRCHECK.

NOGEN is also useful when you are explicitly logging code with CODEGEN.

Example

```
COMPONENT MENU NOTSAVED NOGEN NOERRCHECK;
  TYPE DISCRETE;
  VALUES Main, Cal, A_Meas, B_Meas, Misc, Status, About;
  INITIAL Main;
  SET ACTIONS;
    SELECT MENU;
    CASE Main;
      HIDE A_Meas;
!       .
!       .
!       .
    END SELECT;
END COMPONENT;
```

Cross Reference

- COMPONENT
- NOERRCHECK
- NOPOKEINITIAL
- NOTSAVED

NOPOKEINITIAL

NOPOKEINITIAL is an optional clause that follows the component name in the component description. The NOPOKEINITIAL option instructs the measurement product to not initialize the component during a reset operation.

You should specify the NOPOKEINITIAL option for components that correspond to instrument functions that are not initialized during a reset operation. Examples are status registers and long flatness correction lists.

Example

```
COMPONENT ESREGISTER NOTSAVED NOPOKEINITIAL;
  TYPE INTEGER;
  INITIAL 0;
  GET ACTIONS;
    OUTPUT STRING "*ESR?";
    ENTER ESREGISTER FORMAT K;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

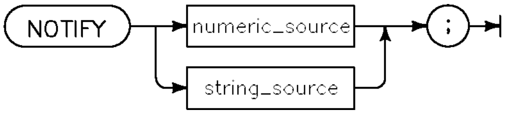
- COMPONENT

NOTIFY

The NOTIFY statement is an action statement. This statement allows you to specify a message that the development environment displays to the user. The development environment then stops executing statements until the user responds by clicking on OK, which is displayed in the dialog box onscreen.

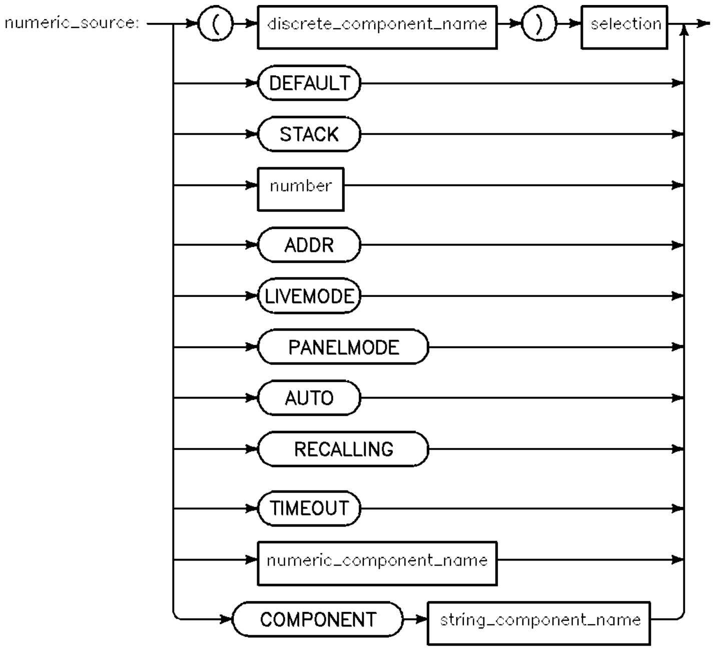
During run time, executing this statement has no effect.

Syntax



Item	Description
numeric_source	See drawing 2.
string_source	See drawing 3.

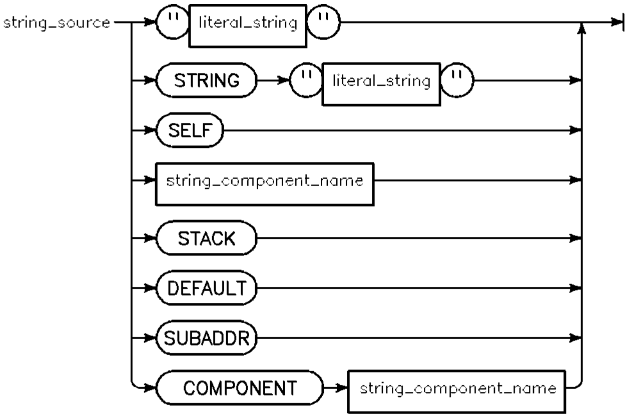
NOTIFY



Drawing 2

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.

NOTIFY



Drawing 3

Item	Description
STRING	The development environment uses a user-entered string as the value.
SELF	The development environment uses the name of the current component as the value.
string_component_name	The development environment uses the name of the STRING component specified.
STACK	The development environment uses the value at the top of the stack.
DEFAULT	The development environment uses the value of the current component.
SUBADDR	The development environment uses the subaddress the user typed into the configuration box.
COMPONENT string	Indicates using a component as the source component name.

Example

```
REVISION 2.0;

COMPONENT START_SCANNER NOTSAVED NOGEN NOERRCHECK;
   TYPE CONTINUOUS;
   PANEL SET ACTIONS;
     OUTPUT STRING "OPT?";
     ENTER START_SCANNER FORMAT "K";
     SELECT START_SCANNER;
       CASE 44491;
         !.
       CASE 44492;
         !.
       CASE 0;
         NOTIFY "No scanner option available";
     END SELECT;
   END ACTIONS;
END COMPONENT;
```

NOTSAVED

NOTSAVED is an optional clause that follows the component name in the component description. The NOTSAVED option instructs the development environment to not include the component when the user stores or recalls a state.

You should specify the NOTSAVED option for components that you do not want to be stored as part of a state and then recalled in a measurement procedure. Typically, components that have only a GET ACTIONS list should be specified as NOTSAVED. Another example of a NOTSAVED component is one that provides a menu that allows the user to go from one subpanel to another.

Example

```
COMPONENT MENU NOTSAVED NOGEN NOERRCHECK;
  TYPE DISCRETE;
  VALUES Main, Cal, A_Meas, B_Meas, Misc, Status, About;
  INITIAL Main;
  SET ACTIONS;
    SELECT MENU;
    CASE Main;
      HIDE A_Meas;
!       .
!       .
!       .
    END SELECT;
END COMPONENT;
```

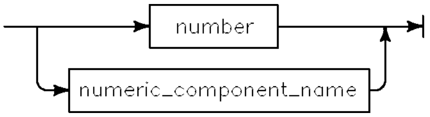
Cross Reference

- COMPONENT
- NOGEN
- NOPOKEINITIAL

numeric_expr

Numeric expressions are used throughout this chapter by statements that need to refer to component values or literal numbers.

Syntax

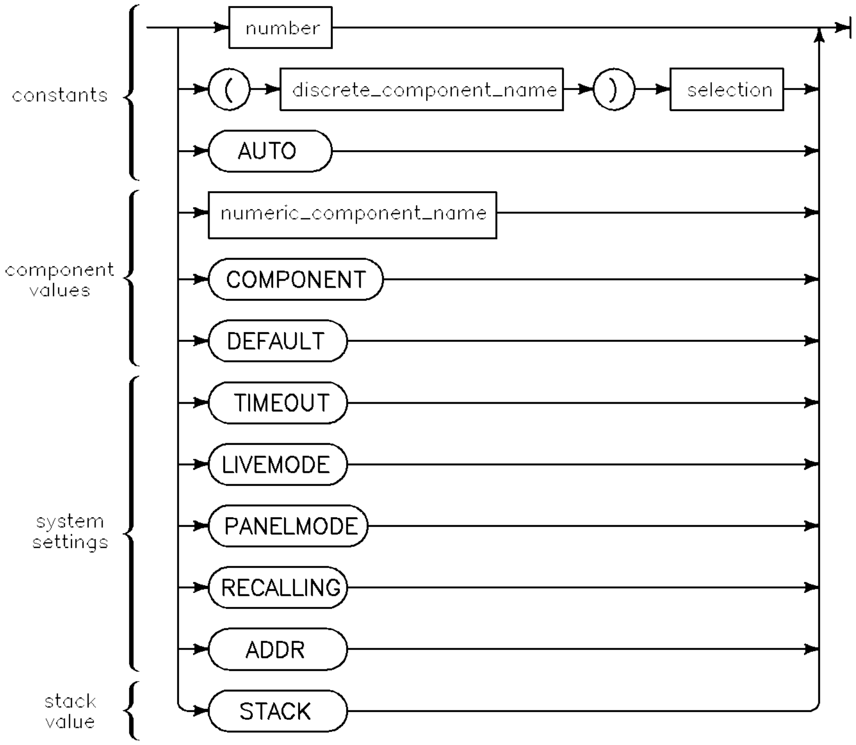


Item	Description
number	An integer or real number.
numeric_component_name	Value of the specified INTEGER or CONTINUOUS component.

numeric_source

Numeric sources are used throughout this chapter to let action statements access numeric component values, constants, system settings, and the stack.

Syntax



Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.

Example

```
FETCH   4.3;
FETCH   (Function) Square;
FETCH   AUTO;
FETCH   Frequency;
FETCH   COMPONENT Frequency;
FETCH   DEFAULT;
FETCH   TIMEOUT;
IF LIVEMODE THEN;  END IF;
IF PANEL MODE THEN; END IF;
IF RECALLING THEN; END IF;
FETCH ADDR;
OUTPUT STACK FORMAT 'K';
```

OUTPUT

The OUTPUT statement is an action statement. It is used to send commands to the instrument.

The development environment buffers outgoing data before it is sent. The following action statements and END ACTIONS cause the development environment to send the buffer and append the EOL sequence to the end.

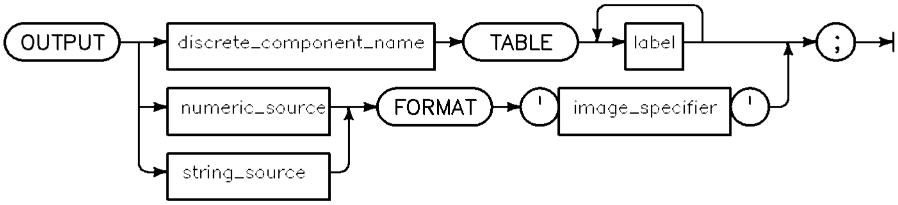
- SET
- GET
- WAIT TIME
- WAIT SPOLL BIT
- ENTER
- TRIGGER
- SPOLL
- USERSUB
- CLEAR
- FLUSH
- POKEINITIAL

A component is validated when it is output. If OUTPUT STACK is used, the value must be numeric.

When OUTPUT TABLE is used, if the value is 0, the first item in the list is sent. If the value is 1, the second item in the list is sent, and so on.

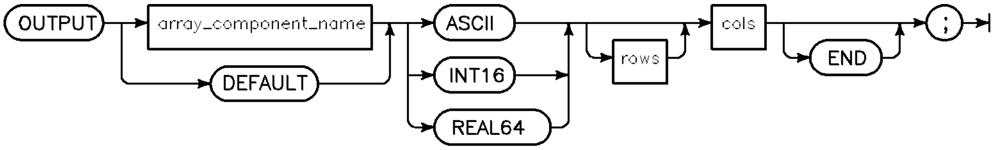
OUTPUT

Syntax



Item	Description
discrete_component_name	Name of a DISCRETE component.
numeric_source	See drawing 3.
string_source	See drawing 4.
label	STRING sent to the instrument depending on the value of the DISCRETE component.
image_specifier	See table 8-11 under this keyword section.

Outputting Arrays



8

Item	Description
array_component_name	Name of an RARRAY or IARRAY component from which the development environment gets the data to output.
DEFAULT	The development environment outputs the value of the component that causes the action list to be executed.
ASCII	Specifies that the development environment will send items separated by commas and terminated with a carriage return/line feed.
INT16	Specifies that the development environment will output 16-bit integer binary data.
REAL64	Specifies that the development environment will output 64-bit IEEE 754 floating point binary data.
rows	A constant or name of a component whose value specifies how many rows of the array to send. If rows is missing, it defaults to 1.
cols	A constant or name of a component whose value specifies how many columns of the array to send.
END	Causes EOI to be sent with the last data byte of the array.

For array outputs, anything in the output buffer is sent over the HP-IB without any EOL processing. Then, the array is sent without any buffering. The output buffer is always empty after an array output. Normal EOL processing is suspended for an array output. EOI is sent if and only if END is specified on the OUTPUT statement.

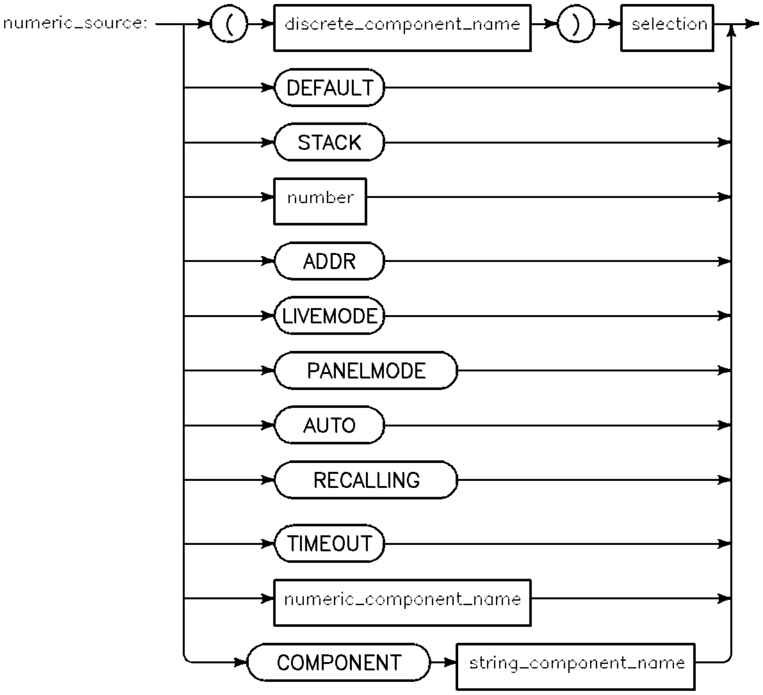
For array outputs, **rows** and **cols** indicate whether they are output with the row or column index incrementing faster. If rows and cols match the rows and cols specified in the IARRAY or RARRAY component statement, then the array is output with the column index incrementing faster. If the rows and cols in the OUTPUT statement are swapped from those specified in the IARRAY or RARRAY component statement, then the array is output with the row index incrementing faster. If rows or cols in the OUTPUT statement is smaller than rows or cols in the array component, then only that many rows or columns are output.

In any case, the smaller of rows or cols in the OUTPUT statement must match rows in the TYPE statement. For the following array declaration:

```
COMPONENT A;

  TYPE IARRAY, 2, 4;
```

Containing data as follows:

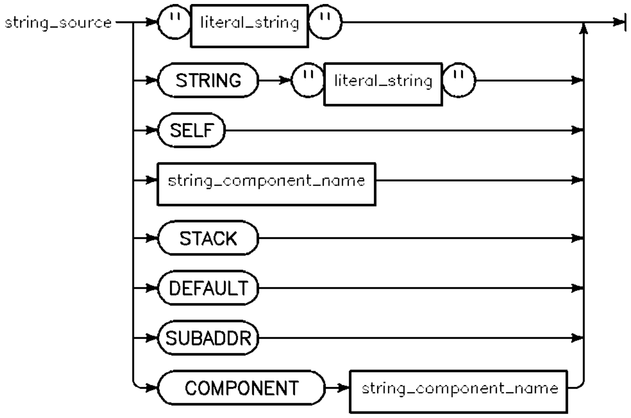
```
1  2  3  4
5  6  7  8

   The statement:         Would output:

OUTPUT A, ASCII, 2, 4    1 2 3 4 5 6 7 8
OUTPUT A, ASCII, 4, 2    1 5 2 6 3 7 4 8
OUTPUT A, ASCII, 3, 2    1 5 2 6 3 7
OUTPUT A, ASCII, 2, 3    1 2 3 5 6 7
```


numeric_source:
(
discrete_component_name
)
selection
DEFAULT
STACK
number
ADDR
LIVEMODE
PANELMODE
AUTO
RECALLING
TIMEOUT
numeric_component_name
COMPONENT
string_component_name

Drawing 3

OUTPUT

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.



Drawing 4

Item	Description
STRING	The development environment uses a user-entered string as the value.
SELF	The development environment uses the name of the current component as the value.
string_component_name	The development environment uses the name of the STRING component specified.
STACK	The development environment uses the value at the top of the stack.
DEFAULT	The development environment uses the value of the current component.
SUBADDR	The development environment uses the subaddress the user typed into the configuration box.
COMPONENT string	Indicates using a component as the source component name.

Table 8-11. Valid OUTPUT FORMAT Specifiers

Specifier	Meaning
K	Compact field. Outputs a number or string in standard form with no leading or trailing blanks.
S	Outputs the number sign (+ or −).
M	Outputs the number's sign if negative, a blank if positive.
D	Outputs 1-digit character. A leading zero is replaced by a blank. If the number is negative and no sign image is specified, the minus sign will occupy a leading digit position. If a sign is printed, it will "float" to the left of the left-most digit.
Z	Same as D, except that leading zeros are output.
.	Outputs a decimal-point radix indicator.
E	Outputs an E, a sign, and a 2-digit exponent.
ESZ	Outputs an E, a sign, and a 1-digit exponent.
ESZZ	Same as E.
ESZZZ	Outputs an E, a sign, and a 3-digit exponent.
A	Outputs a string character.
literal	Outputs the characters in the literal.
B	Outputs a byte. The number is rounded to an INTEGER and the least significant byte is output.

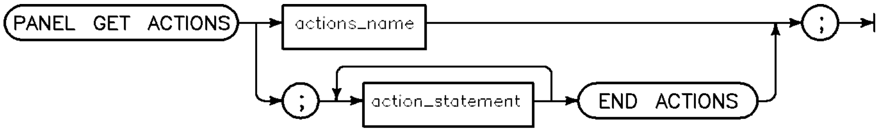
Example

```
REVISION 2.0;

COMPONENT SINGLE;
  TYPE INTEGER;
  SET ACTIONS;
    OUTPUT SINGLE FORMAT "K";
  END ACTIONS;
END COMPONENT;

COMPONENT ARRAY;
  TYPE RARRAY 1 20;
  SET ACTIONS;
    OUTPUT ARRAY ASCII 1 20;
  END ACTIONS;
END COMPONENT;

COMPONENT ONE_OF_N;
  TYPE DISCRETE;
  VALUES ZERO,ONE,TWO;
  SET ACTIONS;
    OUTPUT ONE_OF_N TABLE "FN0","FN1","FN2";
  END ACTIONS
END COMPONENT;
```

PANEL GET ACTIONS

The PANEL GET ACTIONS ... END ACTIONS compound statement allows you to define an action list that is executed only in the development environment. The PANEL GET ACTIONS list should always follow the GET ACTIONS list in a component description. The development environment executes the PANEL GET ACTIONS list after it executes the GET ACTIONS list. The PANEL GET ACTIONS statement is useful for operations that are included only to make the panel work properly but are not necessary to control the instrument in the run-time environment.

Syntax



Item	Description
actions_name	The name of the action list.
action_statement	See tables 8-1 through 8-8 under the ACTIONS keyword for the valid statements.

Cross Reference

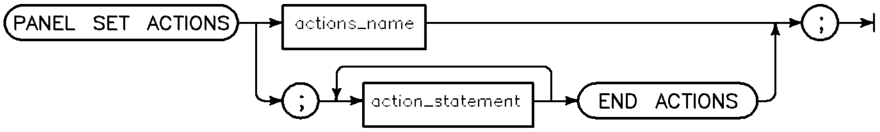
- ACTIONS

8

PANEL SET ACTIONS

The PANEL SET ACTIONS ... END ACTIONS compound statement allows you to define an action list that is executed only in the development environment. The PANEL SET ACTIONS list should always follow the SET ACTIONS list in a component description. The development environment executes the PANEL SET ACTIONS list after it executes the SET ACTIONS list. The PANEL SET ACTIONS statement is useful for operations that are included only to make the panel work properly but are not necessary to control the instrument in the run-time environment.

Syntax



Item	Description
actions_name	The name of the action list.
action_statement	See tables 8-1 through 8-8 under the ACTIONS keyword for the valid statements.

Example

```
REVISION 2.0;

COMPONENT MENU NOTSAVED NOGEN NOERRCHECK;
  TYPE DISCRETE;
  VALUES SETUP, STATUS, RELAY;
  INITIAL SETUP;
  PANEL SET ACTIONS;
    GOSUB HIDE_ALL;
    SELECT MENU;
      CASE SETUP;
        SHOW SETUP_P;
      CASE STATUS;
        SHOW STATUS_P;
      CASE RELAY;
        SHOW RELAY_P;
    END SELECT;
  END ACTIONS;
END COMPONENT;

ACTIONS HIDE_ALL;
  HIDE SETUP_P;
  HIDE STATUS_P;
  HIDE RELAY_P;
END ACTIONS;
```

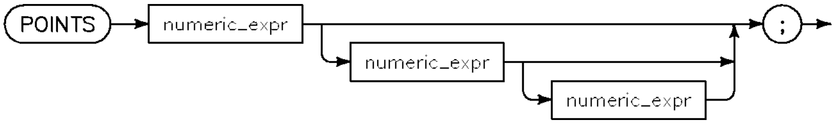
Cross Reference

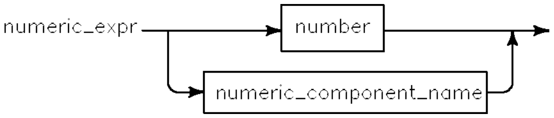
- ACTIONS

POINTS

The POINTS statement allows you to specify the current dynamic dimensions of a trace data type. It is an optional part of the component description. The HP ID writer is responsible for updating the value of this field to reflect the current limits. POINTS has one to three values depending on whether it represents a single or multiple dimensional trace. The total size specified by points cannot exceed the total size specified in the ITRACE or RTRACE statement. The default value for POINTS is the total size declared in the component definition for the first value and one for the second and third values.

Syntax





Item	Description
numeric_expr	Specifies the size currently specified by that dimension of the trace. The product of the three numbers must be less than, or equal to, the dimensioned size.

Example

```
COMPONENT TRACE1;
  TYPE RTRACE 1024;
  TRACETYPE WAVEFORM;
  POINTS WF_SIZE;
  XMIN X_MIN;
  XINCR X_INCR;
  XLOG OFF;
  XUNIT "s";
  YUNIT "V";

COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;

COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- COMPONENT

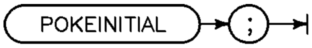
POKEINITIAL

The POKEINITIAL statement is an action statement. This statement sets the value of all components in the HP ID file to their initial value and initial state, as specified by the INITIAL statement.

When the development environment executes the POKEINITIAL statement, it sets the values of the appropriate components but doesn't update the instrument.

This statement has no parameters.

Syntax



Example

```
REVISION 2.0;
INITIALIZE COMPONENT RESET; ! Executes SET ACTIONS of component RESET.
!     .
!     .
!     .
COMPONENT UNITS;
  TYPE DISCRETE;
  VALUES LOG,LINEAR;
  INITIAL LOG;   ! Sets UNITS to LOG when POKEINITIAL executed.
!     .
!     .
!     .
END COMPONENT;
!
```

(Continued)

```
COMPONENT A_CAL_ADJUST;
  TYPE CONTINUOUS;
  VALUES RANGE 50,120,.1;
  INITIAL 100  ! Sets A_CAL_ADJUST to 100 when POKEINITIAL executed.
END COMPONENT;
!
COMPONENT A_CAL NOTSAVED;
  TYPE INTEGER;  ! Defaults to initial value of 0.
  SET ACTIONS;
    OUTPUT A_CAL_ADJUST FORMAT '"AECL",K,"ENEN"';
  END ACTIONS;
END COMPONENT;
!
COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
    CLEAR;
!     .
!     .
!     .
    POKEINITIAL; ! Sets components to initial value.
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- INITIAL
- INITIALIZE COMPONENT
- NOPOKEINITIAL

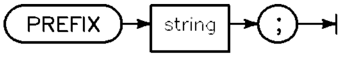
PREFIX

The PREFIX statement specifies a string that the development environment sends to the instrument before it executes any SET ACTIONS or GET ACTIONS lists. This is useful in formatting a card-cage instrument's subaddress.

The PREFIX statement enables the development environment to address card-cage instruments that require addressing information at the beginning of the data stream.

Each time the PREFIX statement is executed, the development environment formats the subaddress field as specified by the PREFIX statement. This is the field in the instrument configuration box that the user enters a subaddress into when working in the development environment. If this field is blank, the PREFIX is not sent.

Syntax



Item	Description
string	See table 8-11 for specifiers (under OUTPUT).

Example

```
REVISION 2.0;

PREFIX '"USE ",K,";"';

COMPONENT A;
    TYPE INTEGER;
    SET ACTIONS;
        OUTPUT A FORMAT '"SET A ",K';
    END ACTIONS;
END COMPONENT;

PANEL MAIN;
    POSITION 1,1;
    SIZE 200,200;

    CONTINUOUS A;
        POSITION 10,10;
        TITLE "A";
    END CONTINUOUS;
END PANEL;
```

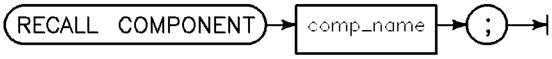
RECALL COMPONENT

RECALL COMPONENT is a general HP ID statement that allows you to specify a component whose SET ACTIONS are executed during a state recall.

The component must be of TYPE INTEGER. During a state recall, after the development environment has put all of the desired values into the components, the SET ACTIONS of the RECALL COMPONENT will be executed.

In addition, during the recall the value of the component specified by RECALL COMPONENT will be set to 1. At all other times, the RECALL COMPONENT will be 0.

Syntax



Item	Description
comp_name	The name of an INTEGER component whose SET ACTIONS you want executed during a state recall.

Example

```
REVISION 2.0;

STORE COMPONENT STORE_STATE;
RECALL COMPONENT LEARNSTRING;

COMPONENT LEARNSTRING;
        TYPE IARRAY 1,512;
        SET ACTIONS;
                OUTPUT "LEARN";
                OUTPUT LEARNSTRING INT16 1 512 END;
                VALIDATE ALL;
        END ACTIONS;
END COMPONENT;

COMPONENT STORE_STATE NOTSAVED;
        TYPE INTEGER;
        SET ACTIONS;
                OUTPUT "LEARN?";
                ENTER LEARNSTRING INT16 1 512;
        END ACTIONS
END COMPONENT;

PANEL MAIN;
   SIZE 200,200;
   POSITION 1,1;
END PANEL;
```

Cross Reference

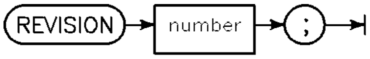
- STORE COMPONENT

REVISION

The REVISION statement indicates the revision level of the driver standard to which the HP ID is written.

The REVISION statement must be the first non-comment statement in the HP ID file.

Syntax



Item	Description
number	A real number that specifies the version of the HP ID. It must be 2.0 for HP IDs written using the specifications described in this manual.

Example

```
!Opening comment lines

REVISION 2.0;
!
INITIALIZE COMPONENT RESET;
ERROR COMPONENT ERROR;
!      .
!      .
```

SELECT ... END SELECT

SELECT ... END SELECT is a compound action statement. It is similar to the IF ... END IF compound action statement, however it allows you to define several conditional action lists instead of one or two. The development environment executes only one action list each time the statement is executed. Each action list begins after a CASE or CASE ELSE statement, and ends when the next program line is a CASE, CASE ELSE, or END SELECT statement.

The SELECT statement specifies a source (see drawing 1, following page) whose value is compared to the list of values in each CASE statement. When a match is found, the development environment executes the corresponding action list. The remaining action lists are skipped, and the development environment continues execution with the first program line following the END SELECT statement.

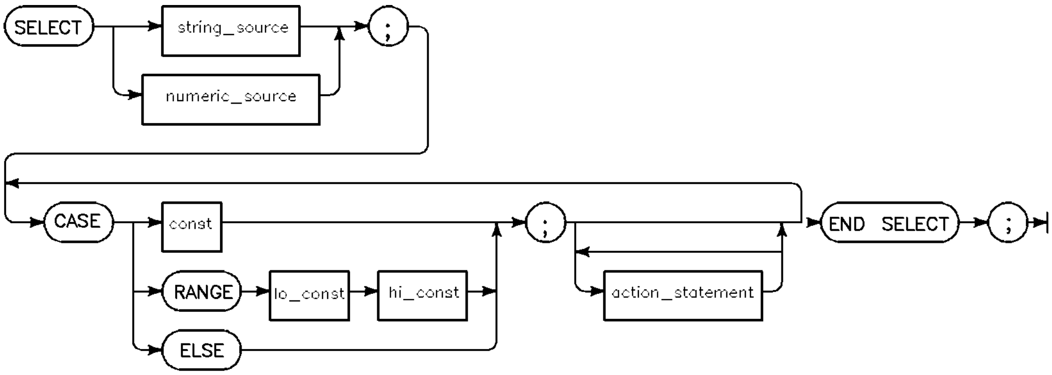
All CASE values must be of the same type (for example, STRING, DISCRETE, and so on).

The optional CASE ELSE statement defines an action list that is to be executed when the SELECT statement value does not match any CASE statement value.

You can nest IF ... END IF, LOOP ... END LOOP, and SELECT ... END SELECT compound statements up to ten deep.

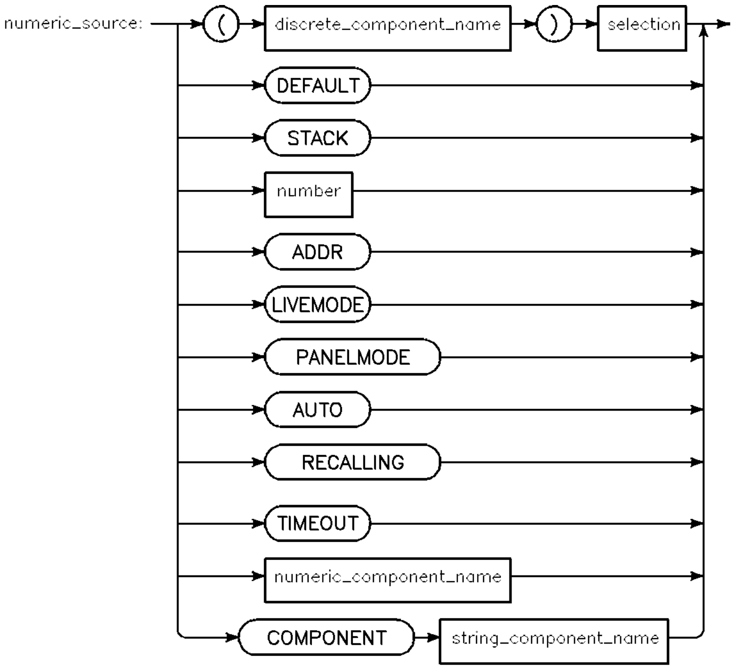
The development environment requires that either a CASE ELSE or a CASE for each possible value be included.

Syntax



Item	Description
numeric_source	See drawing 2.
string_source	See drawing 3.
const	A numeric or string value that the development environment compares with the value provided by the source. If they are the same, then the development environment executes the action list that follows CASE const. If the source is a DISCRETE component, the value can be of the form **(comp_name) discrete_selection**.
RANGE	Only valid if a **numeric_source** is used. If the value provided by the source is within the numbers specified in **lo_const** and **hi_const**, the development environment executes the action list that follows CASE RANGE.
lo_const	The lowest numeric value in a RANGE.
hi_const	The highest numeric value in a RANGE.
action_statement	See tables 8-1 through 8-8 under the ACTIONS keyword for the valid statements.

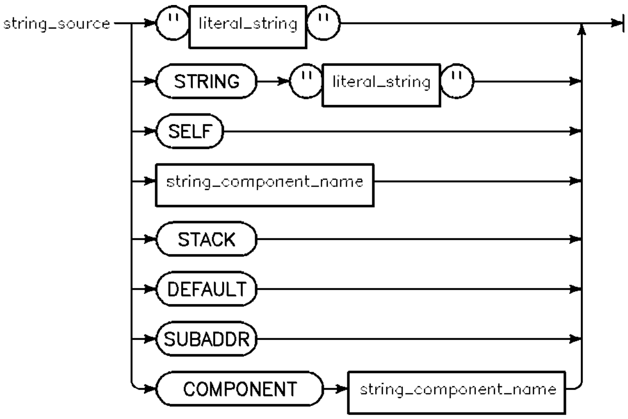
SELECT . . . END SELECT



Drawing 2

8

Item	Description
(discrete_component_name) selection	The selection of the specified DISCRETE component.
DEFAULT	Value of the current INTEGER or CONTINUOUS.
STACK	Value at the top of the stack.
number	An integer or real number.
ADDR	Address specified in the configuration box.
LIVEMODE	"1" if LIVE MODE is on, otherwise, "0".
PANELMODE	"1" if running in the development environment, otherwise, "0".
AUTO	The value AUTO may be used if the current component is CONTINUOUS and its VALUES statement specifies AUTO.
RECALLING	"1" if currently performing a state recall, otherwise, "0".
TIMEOUT	Timeout for the device specified in the configuration box.
numeric_component_name	Value of the specified CONTINUOUS or INTEGER component.
COMPONENT numeric_component_name	Value of the specified CONTINUOUS or INTEGER component. Usually used when working with ambiguous component names.



Drawing 3

Item	Description
STRING	The development environment uses a user-entered string as the value.
SELF	The development environment uses the name of the current component as the value.
string_component_name	The development environment uses the value of the STRING component specified.
STACK	The development environment uses the value at the top of the stack.
DEFAULT	The development environment uses the value of the current component.
SUBADDR	The development environment uses the subaddress the user typed into the configuration box.
COMPONENT string	Indicates using a component as the source component name.

Example

```
REVISION 2.0;

COMPONENT RANGE;
    TYPE CONTINUOUS;
    VALUES RANGE 1 5 AUTO;
    SET ACTIONS;
        SELECT RANGE;
            CASE AUTO;
                OUTPUT STRING "RA";
            CASE RANGE 0,1;
                OUTPUT STRING "R0";
            CASE RANGE 1,2;
                OUTPUT STRING "R1";
            CASE RANGE 2,3;
                OUTPUT STRING "R2";
            CASE RANGE 3,4;
                OUTPUT STRING "R4";
        END SELECT;
    END ACTIONS;
END COMPONENT;

PANEL MAIN;
    POSITION 1,1;
    SIZE 200,200;

    CONTINUOUS RANGE;
        POSITION 70,40;
        TITLE "Range";
    END CONTINUOUS;
END PANEL;
```

Cross Reference

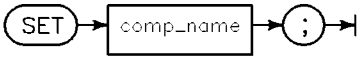
- IF . . . END IF

SET

The SET statement is an action statement. When the SET statement is executed in an action list, the development environment executes the SET ACTIONS of the component specified in the statement, and then the PANEL SET ACTIONS if you are in the development environment.

If the component being set is COUPLED to another, either a GET or an INVALIDATE is done on each component in this component's COUPLED list (see COUPLED).

Syntax



Item	Description
comp_name	The name of the component whose SET ACTIONS are executed when the SET statement is executed in an action list.

Example

```
REVISION 2.0;

COMPONENT A;
    TYPE INTEGER;
    SET ACTIONS;
        OUTPUT A FORMAT ’"SET A ",K’;
    END ACTIONS;
END COMPONENT;

ACTIONS RESET_A;
    FETCH 0;
    STORE A;
    SET A;
END ACTIONS;

PANEL MAIN;
    SIZE 200,200;
END PANEL;
```

Cross Reference

- COUPLED

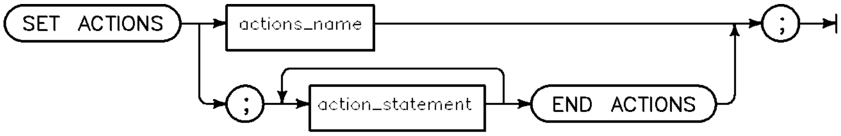
SET ACTIONS

The SET ACTIONS ... END ACTIONS compound statement defines an action list that will SET the value of a component. Usually the SET ACTIONS will send the value of the component to the instrument.

SET ACTIONS will be executed when:

- The SET instruction references this component in an action list.
- hpt_set, hpt_set_str, hpt_set_ary, or hpt_set_iary is executed on this component.
- hpt_set2, hpt_set_str2, hpt_set_ary2, or hpt_set_iary2 is executed on this component.

Syntax



Item	Description
actions_name	The name of the action list.
action_statement	See tables 8-1 through through 8-8 under the ACTIONS keyword for the valid statements.

Example

```
REVISION 2.0;

ACTIONS FREQ_0;
   FETCH 0;
   STORE FREQ;
END ACTIONS;

COMPONENT DC;
  TYPE INTEGER;
  SET ACTIONS;
    OUTPUT STRING "DC";
    GOSUB FREQ_0;
  END ACTIONS;
END COMPONENTS;
```

Cross Reference

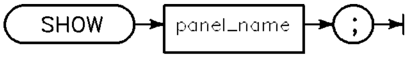
- ACTIONS

SHOW

The SHOW statement is an action statement. This statement allows you to display a subpanel.

All the subpanels of the panel specified are also shown, unless they have been hidden with the HIDE statement.

Syntax



Item	Description
panel_name	The name of the subpanel that you want displayed with the parent panel.

Example

```
COMPONENT RESET;
  TYPE INTEGER;
  SET ACTIONS;
!    .
!    .
!    .
  END ACTIONS;
  PANEL SET ACTIONS;
    SHOW TOP;
    HIDE WATT_READING;
    INVALIDATE A_RANGE;
  END ACTIONS;
END COMPONENT;
```

(Continued)

```
!      .
!      .
!      .
!Panel section
PANEL HP_438;
  PANEL TOP;
    POSITION 0,182;
!      .
!      .
!      .
  END PANEL;
  PANEL WATT_READING;
    POSITION 10,150;
    PANEL SUB_WATT;
!      .
!      .
!      .
    END PANEL;
  END PANEL;
!      .
!      .
!      .
END PANEL;
```

Cross Reference

- HIDE

SKIP EOL

The SKIP EOL statement is an action statement. It is used to suppress the EOL processing that would occur the next time the development environment sends a command to the HP-IB. In general, each time the development environment sends a command to the HP-IB, it sends the current EOL setting after the command, which, by default, would append a carriage return and line feed to the data being sent.

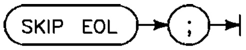
If SKIP EOL is in effect, the string is sent to the HP-IB, but no EOL characters or EOI is sent.

If a SKIP EOL statement is executed, it is in effect until just after one of the following:

- A FLUSH statement is executed.
- An END ACTIONS statement is executed unless the END ACTIONS statement was reached by executing a GOSUB statement.

If SKIP EOL is in effect and another SET or GET is done, EOL processing will be restored while executing that SET or GET action list, but once back in the original action list, SKIP EOL will again take effect.

Syntax



Example

```
REVISION 2.0;

COMPONENT A;
        TYPE INTEGER;
        SET ACTIONS;
                GOSUB S;
                OUTPUT...
                ENTER...      ! Implicit flush without EOL
                SET B;
                OUTPUT...
                FLUSH         ! This explicit FLUSH outputs
                              ! the string without EOL,
                              ! but then turns EOL processing
                              ! back on.
                OUTPUT...
        END ACTIONS;          ! Implicit flush with EOL
END COMPONENT;

COMPONENT B;
        TYPE INTEGER;
        SET ACTIONS;
                OUTPUT...
                ENTER...      ! Implicit flush with EOL
        END ACTIONS;
END COMPONENT;

ACTIONS S;
        SKIP EOL;
        OUTPUT...
        ENTER...              ! Implicit flush without EOL
END ACTIONS;
```

SKIP EOL

In this example, suppose a

```
SET A;
```

is executed. The development environment executes GOSUB S, which causes the development environment to execute SKIP EOL. EOL processing is now off. Any strings sent to the HP-IB will not have EOL commands. Returning from S, EOL processing is still off, since this END ACTIONS statement was reached by executing a GOSUB.

The next OUTPUT/ENTER sequence (in component A's SET ACTIONS list) has no EOL processing. The SET B, however, causes development environment to execute B's SET ACTIONS with EOL processing on. Once we return to A's SET ACTIONS, EOL processing is again off. The FLUSH then turns EOL processing on.

Cross Reference

- FLUSH
- EOL

SKIP ERRCHECK

The SKIP ERRCHECK statement is an action statement. It is used to suppress the error checking normally done by the development environment when error checking mode is on.

If error checking mode is on, the development environment will check for an error by executing the GET ACTIONS of the component specified by the ERROR COMPONENT just before an hpt_set or hpt_get call is completed.

Note that the action statement:

```
    SET comp_name;
```

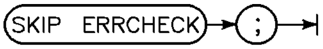
does not cause another call to hpt_set and, therefore, does not invoke error handling at the end of the comp_name SET ACTIONS list.

If the SKIP ERRCHECK statement is executed while executing an action list directly as a result of an hpt_set or hpt_get, the error check that would have occurred at the end of the hpt_set or hpt_get will not be done even if error checking mode is on.

If the SKIP ERRCHECK statement is executed while executing an action list invoked by a SET or GET keyword, no error checking would have occurred anyway, and the SKIP ERRCHECK statement has no effect.

SKIP ERRCHECK is useful if your HP ID has set up the instrument for a measurement to be taken later, and the error query would disturb the instrument setup.

Syntax



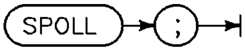
Cross Reference

- ERROR COMPONENT

SPOLL

The SPOLL statement is an action statement. The SPOLL statement performs a serial poll of the instrument and pushes the result on the stack.

Syntax



Example

```
COMPONENT SPOLL_VALUE NOTSAVED;
  TYPE INTEGER;
  GET ACTIONS;
    SPOLL;
    STORE SPOLL_VALUE;
  END ACTIONS;
END COMPONENT;
```

STORE

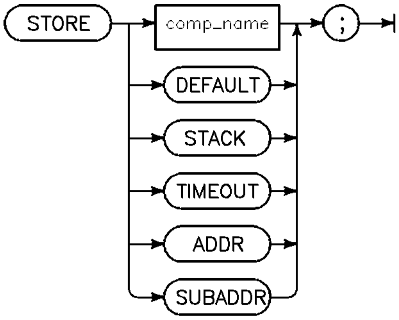
The STORE statement is an action statement. This statement along with the FETCH statement and the development environment arithmetic, logical, and string operators (tables 8-1 through 8-8 under the ACTIONS keyword) form the vocabulary of a stack machine.

The STORE statement pops the value from the top of the stack and places it into the specified destination.

Note

The component is VALIDATEd unless its state is DONTCARE.

Syntax



STORE

Item	Description
comp_name	The value at the top of the stack replaces the value of component comp_name.
DEFAULT	The value at the top of the stack replaces the value of the component whose action list contains the STORE statement.
STACK	The value at the top of the stack replaces the value at the top of the stack. This is a no-op.
TIMEOUT	The value at the top of the stack replaces the timeout of the device (usually specified when configuring the device.
ADDR	The value at the top of the stack replaces the address of the device (usually specified when configuring the device).
SUBADDR	The value at the top of the stack replaces the subaddress of the device (usually specified when configuring the device).

Storing Into an Array

You can store a number into a single element of an array. This is done by pushing the value, then the indexes, onto the stack, then storing to comp_name (see FETCH). To store a value into an array, use the following statements in an action list:

- **One-dimension array:**
  - □ FETCH value
  - □ FETCH index
  - □ STORE array_comp_name
- **Two-dimension array:**
  - □ FETCH value
  - □ FETCH row_index
  - □ FETCH col_index
  - □ STORE comp_name

Example

```
FETCH SRQ_MASK; ! Puts value on stack
FETCH 0; ! Puts value on stack
BIT; ! Puts the 0-position bit of SRQ_MASK on stack
STORE SRQ_FAULT; ! and stores it in SRQ_FAULT
```

STORE COMPONENT

STORE COMPONENT is a general HP ID statement. A STORE COMPONENT may be declared in an HP ID if you need to alter the behavior of the development environment when a state is stored.

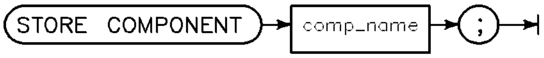
When a state is stored, the component specified in the STORE COMPONENT is SET to a 1. The component must be TYPE INTEGER.

If the instrument is so complex that it is difficult to store and recall states as a combination of storing and recalling component values, perhaps the instrument state can be stored and recalled some other way.

For example, the instrument learn string can be used to store and recall the state of some complex instruments.

The SET ACTIONS of the component specified in the STORE COMPONENT can then query the instrument for its learn string, and then another component can send this learn string to the instrument when a state recall occurs.

Syntax



Item	Description
comp_name	The name of an INTEGER component whose SET ACTIONS you want executed during a store state.

Example

```
REVISION 2.0;

STORE COMPONENT STORE_STATE;
RECALL COMPONENT LEARNSTRING;

COMPONENT LEARNSTRING;
        TYPE IARRAY 1,512;
        SET ACTIONS;
                OUTPUT "LEARN";
                OUTPUT LEARNSTRING INT16 1 512 END;
                VALIDATE ALL;
        END ACTIONS;
END COMPONENT;

COMPONENT STORE_STATE NOTSAVED;
        TYPE INTEGER;
        SET ACTIONS;
                OUTPUT "LEARN?";
                ENTER LEARNSTRING INT16 1 512;
        END ACTIONS;
END COMPONENT;

PANEL MAIN;
   SIZE 200,200;
   POSITION 1,1;
END PANEL;
```

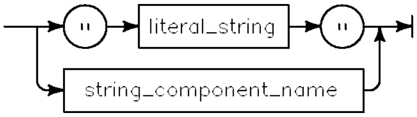
Cross Reference

- RECALL COMPONENT

string_expr

String expressions are used throughout this chapter by statements that need to refer to component values or literal strings.

Syntax

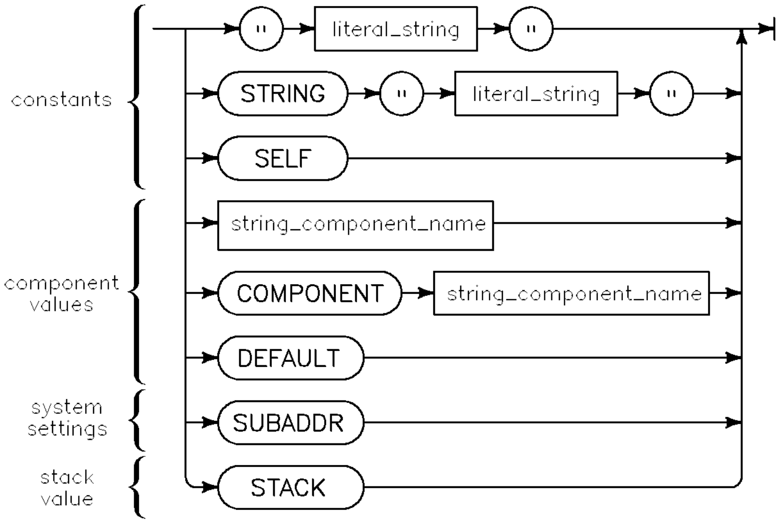


Item	Description
"literal string"	An actual string enclosed in double quotes.
string_component_name	Value of the specified string component.

string_source

String sources are used throughout this chapter to let action statements access string component values, constants, system settings, and the stack.

Syntax



string_source

Item	Description
"literal string"	An actual string enclosed in double quotes.
STRING "literal string"	An actual string enclosed in double quotes. This syntax can be used to avoid certain language ambiguities.
SELF	Name of the current component.
string_component_name	Value of the specified STRING component.
COMPONENT string_component name.	Value of the specified STRING component. This syntax can be used when dealing with ambiguous component names.
DEFAULT	Value of the current component.
SUBADDR	Subaddress of the device (usually specified when configuring the device.
STACK	Value at the top of the stack.

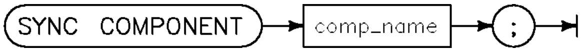
SYNC COMPONENT

SYNC COMPONENT is a general HP ID statement that allows you to specify a component whose GET ACTIONS are executed during a sync operation.

The component must be of TYPE INTEGER. During a sync operation, the development environment will execute the GET ACTION of the specified component. This allows the HP ID to synchronize its state with the current instrument state.

In addition, during the synchronization, the value of the component specified by the SYNC COMPONENT will be set to 1. At all other times, the SYNC COMPONENT will be 0.

Syntax



Item	Description
comp_name	The component whose GET ACTIONS are executed during a SYNC operation. This must be a TYPE INTEGER or CONTINUOUS component.

Example

```
REVISION 2.0;

SYNC COMPONENT SYNC_STATE;

COMPONENT SYNC_STATE NOTSAVED;
  TYPE INTEGER;
  GET ACTIONS;
    GET Freq_cent;
    GET Freq_span;
    ...
  END ACTIONS;
END COMPONENT;
```

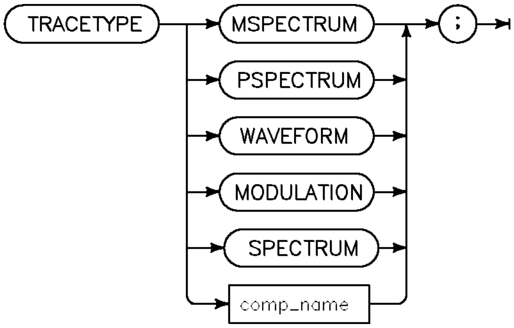
TRACETYPE

The TRACETYPE statement allows you to specify the current dynamic dimensions of an array or trace data type. It is an optional part of the component description and must follow the COUPLED statement.

The HP ID writer is responsible for updating the value of this field to reflect the current limits.

Value	Tracetype
0	NOT USED
1	MSPECTRUM
2	PSPECTRUM
3	WAVEFORM
4	MODULATION
5	SPECTRUM

Syntax



Item	Description
MSPECTRUM	Magnitude spectrum. The X axis of the trace is in the frequency domain and the Y axis values represent signal magnitude.
PSPECTRUM	Phase spectrum. The X axis of the trace is in the frequency domain and the Y axis values represent signal phase.
WAVEFORM	The X axis of the trace is in the time domain and the Y axis values represent signal magnitude.
MODULATION	The X axis of the trace is in the time domain and the Y axis values represent frequency.
SPECTRUM	A trace of complex values. The X axis is in the frequency domain and the Y axis values are an ordered pair of real and imaginary magnitude.

Example

```
COMPONENT TRACE1;
  TYPE RTRACE 1024;
  TRACETYPE WAVEFORM;
  POINTS WF_SIZE;
  XMIN X_MIN;
  XINCR X_INCR;
  XLOG OFF;
  XUNIT "s";
  YUNIT "V";

COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;
```

(Continued)

TRACETYPE

```
COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

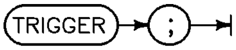
Cross Reference

- COMPONENT

TRIGGER

The TRIGGER statement is an action statement. The TRIGGER statement causes the development environment to send an HP-IB Group Execute Trigger (GET) command to the instrument.

Syntax

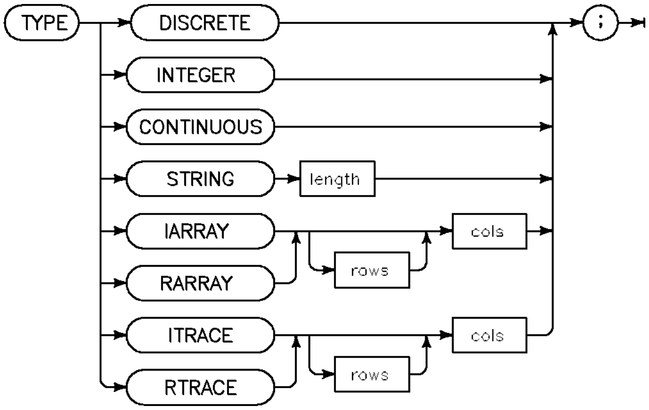


TYPE

The TYPE statement is a required part of the component description. This indicates to the development environment what kind of component this is.

The keyword TYPE must be followed by one of eight possible types.

Syntax



Item	Description
DISCRETE	Used to represent instrument controls that have 1-of-n values, such as the Function control on a multimeter, which has a list of settings. Be cautious about creating a 1-of-n list for a control that is not really designed that way (for example, attenuator settings of 0 or 10 dB) because an hpt_set_str and hpt_get_str are used with DISCRETE components and they require strings. Hence, the value will be treated as a string.
INTEGER	Used to represent values that are inherently whole numbers (for example, an SRQ mask).
CONTINUOUS	Used to represent real numbers or values outside of the range of 16-bit integers.
STRING length	Used to represent strings. Requires that you specify the length of the string. The string may be up to 256 characters long.
IARRAY	Used to represent integer array data.
RARRAY	Used to represent real array data, such as a waveform.
ITRACE	Used to represent integer trace data.
RTRACE	Used to represent real trace data.
rows	Optional integer that specifies the number of rows in the array. Defaults to 1.
cols	Required integer that specifies the number of columns in the array. Must be greater than or equal to rows in order for XY displays and array I/O to work correctly.

Example

```
REVISION 2.0;

COMPONENT INT;
    TYPE INTEGER;
END COMPONENT;

COMPONENT REAL;
    TYPE CONTINUOUS;
END COMPONENT;

COMPONENT STR;
    TYPE STRING 20;
END COMPONENT;

COMPONENT IA;
    TYPE IARRAY 1 20;
END COMPONENT;

COMPONENT RA;
    TYPE RARRAY 1 20;
END COMPONENT;

COMPONENT DISC;
    TYPE DISCRETE;
    VALUES A,B,C;
END COMPONENT;

PANEL MAIN;
END PANEL;
```

Cross Reference

- COMPONENT

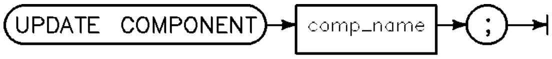
UPDATE COMPONENT

The UPDATE COMPONENT is an HP ID statement that lets you specify a component whose GET ACTIONS will be executed repetitively as long as no other actions lists are being executed, and there is no cursor movement in the development environment.

The UPDATE COMPONENT must be TYPE INTEGER. If Automatic Update is turned on in the development environment, then UPDATE COMPONENT will initiate an update to a specified component or set of components. This is done by setting the value of the UPDATE COMPONENT to 1 then executing the SET ACTIONS of the UPDATE COMPONENT.

If Automatic Update is turned off in the development environment, then the UPDATE COMPONENT will finish the update so that the device will respond to other commands the development environment may send over the bus. This is done by setting the value of the UPDATE COMPONENT to a 0 and then executing the SET ACTIONS of the UPDATE COMPONENT.

Syntax



For a simple HP ID where the device can execute the actions for the UPDATE COMPONENT quickly, the UPDATE COMPONENT can be written as follows:

Example

```
COMPONENT UPDATE_READING;
   TYPE INTEGER;
   INITIAL 0;
   GET ACTIONS;
      GET READING;
   END ACTIONS;
END COMPONENT;

UPDATE COMPONENT UPDATE_READING;
```

For many HP IDs, however, this component may take several seconds to execute, slowing down the response time. Therefore, it is recommended that the UPDATE COMPONENT be written so that each time the GET ACTIONS are executed, a minimal amount of time is spent in that component, that is, the GET READING should be broken into several pieces or phases. The following example shows this technique:

```
COMPONENT UPD_PHASE:  ! Storage place for the current phase of the update.
   TYPE INTEGER;
   INITIAL 1;
END COMPONENT;

! Up_d reading divided into 3 phases: trigger reading, poll to see if
!      data is ready, and enter the data.
! Component is NOTSAVED since an update should be restarted if a
!      state is recalled.  Component is NOERRCHECK since an error
!      check in the middle of the reading will cause the device to
!      hang.
COMPONENT UPD_READING NOTSAVED NOERRCHECK;
  TYPE INTEGER;
  SET ACTIONS;
     SELECT DEFAULT;
!
  CASE 0; ! Finish the update.
        LOOP;
            FETCH UPD_PHASE; ! Get the current phase
            FETCH 1;
            SUB;
            NOT; ! Is current phase=1 (ready to start new reading)
            EXIT IF STACK; ! (exit if current phase=1)
            FETCH LIVEMODE; ! (is the instrument live?)
            NOT;
            EXIT IF STACK; ! (exit if livemode is disabled)
            FETCH 1; ! otherwise, do SET ACTIONS of upd_reading.
                     ! value must be 1 to finish the next phase.
            STORE UPD_READING;
            SET UPD_READING; ! do the next phase of the update.
            FETCH 0; ! re-store the value of this component.
            STORE UPD_READING;
        END LOOP;
```

(Continued)

```
!
     CASE 1;
     GET UPD_READING; ! initiate a reading.
!
     END SELECT;
  END ACTIONS;
!
  GET ACTIONS;
     SELECT UPD_PHASE;
       CASE 1; ! phase 1: trigger a reading
          TRIGGER;
          FETCH 2; ! indicates that phase 2 is to be done next time
          STORE UPD_PHASE;
!
       CASE 2; ! phase 2: SPOLL for data ready to be sent
          SPOLL;
          FETCH 1;
          BINAND; ! If BIT 0 is set, data is ready.

          IF STACK THEN;
             FETCH 3; ! indicates the phase 3 is to be done next time
             STORE UPD_PHASE;
          !ELSE must repeat this phase next time
          END IF;
!
       CASE 3; ! phase 3: get the data
          ENTER READING FORMAT K; ! enter data into desired component
          FETCH 1; ! indicates that phase 1 is to done next time
          STORE UPD_PHASE;
!
    END SELECT;
!
  END ACTIONS;
!
END COMP;
!
UPDATE COMPONENT UPD_READING;
```

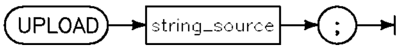
UPLOAD

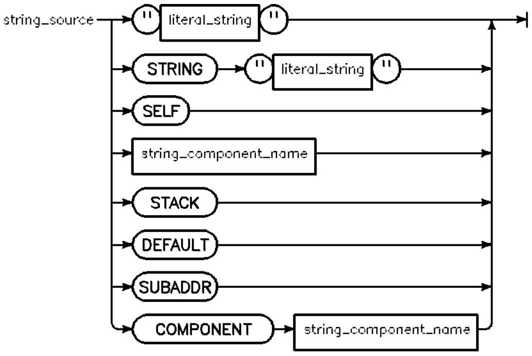
This statement causes the data ready for output from an instrument to be transferred to a file on host computer.

The file is transferred byte for byte and the development product performs no data processing function.

The file name is specified in the string source and is interpreted by the file system of the host computer.

Syntax





Item	Description
string_source	The name of the file to be uploaded to the instrument.

Example

```
COMPONENT SEND_HOP-LIST NOTSAVED;
  TYPE STRING 32;
  SET ACTIONS;
    FETCH SEND_HOP_LIST;
    FETCH "";
    NE;
    IF STACK THEN;
      OUTPUT STRING "HOP:LIST?"
      UPLOAD SEND_HOP_LIST;
    END IF;
  END ACTIONS;
END COMPONENT;

...

PANEL HOP;
  ...
  INPUT SEND_HOP_LIST;
    POSITION 4,55;
    SIZE 120,19;
    STYLE "FILESELECT"";
  END INPUT;
```

Cross Reference

- DOWNLOAD

USERSUB (HP ITG only)

Caution This keyword will work only with the HP ITG for HP BASIC (product number HP E2000A).

The USERSUB statement is an action statement. This statement is used to call an HP BASIC subprogram. It is not used to call HP ITG subprograms.

This statement should only be used to call subprograms that perform actions not provided by the other HP ITG action statements.

Caution You should be cautious about developing HP IDs that call subprograms because Hewlett-Packard cannot guarantee that all future revisions of HP ITG will accommodate the subprograms.

When you use this statement, HP ITG passes a set of parameters to the subprogram in a named common block called /hpt_app/. The common block contains HP ITG's internal number for the instrument. HP ITG establishes the internal number when the user adds it to the soft test system in the development environment. HP ITG's internal number identifies the component that contains the action list that includes the USERSUB statement. The component also contains a value for app_op indicating one of the following operations:

If app_op = 1 The component's HIT ACTIONS or UPDATE ACTIONS are being executed.

If app_op = 4 the component's SET ACTIONS are being executed.

If app_op = 6 hpt_set is being executed.

If app_op = 7 the component's GET ACTIONS are being executed.

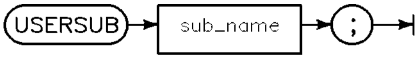
If app_op = 8 the panel's Query button has been clicked.

If app_op = 9 hpt_get is being executed.

If app_op = 10 hpt_recall is being executed.

User subprograms are useful when an instrument returns readings in a packed notation that cannot be decoded using format specifiers. They are also useful for developing applications. See the Application ADFREQRESP provided with the HP ITG driver software for an example.

Syntax



Item	Description
sub_name	The name of an HP BASIC subprogram.

Example

The component ANA_CHANGED is used to communicate to subprogram Hp3852_delrange what has changed.

```
COMPONENT ANA_CHANGED NOTSAVED NOERRCHECK;
  TYPE INTEGER;
  INITIAL 0;
END COMPONENT;

ACTIONS FUNC_ACT;
   FETCH 0;
   STORE ANA_CHANGED;

   SELECT DVM_RANGE;
   CASE AUTO;
   CASE ELSE;
     FETCH AUTO;
     STORE DVM_RANGE;
     FETCH 1;
     STORE ANA_CHANGED;
   END SELECT;

   SELECT DELAY;
   CASE AUTO;
   CASE ELSE;
     FETCH 2;
     FETCH ANA_CHANGED;
     ADD;
     STORE ANA_CHANGED;
   END SELECT;

   IF PANELMODE THEN;
     USERSUB HP3852_DELRANGE;
   END IF;
END ACTIONS;
```

This HP ID calls the subprogram on the following page.

```
SUB Hp3852iv_init
SUBEND

! The subprogram Hp3852_delrange tells the user whether
! delay and range have changed.

SUB Hp3852_delrange
COM /Hpt_app/ INTEGER App_dev,App_comp,App_op
  REAL Ana_changed
    DIM Line1$[25],Line2$[25]
    Hpt_peek(App_dev,"ANA_CHANGED",Ana_changed)
     Line2$="      reset to AUTO."
     SELECT Ana_changed
     CASE <1.5
       Line1$="      Range has been"
     CASE <2.5
       Line1$="      Delay has been"
     CASE ELSE
       Line1$="      Range & Delay have"
       Line2$="      been reset to AUTO"
     END SELECT
     Hpt_dlgnotify1("NOTE: DVM options changed",Line1$,Line2$)
  SUBEND
```

The HP BASIC Subprogram

VALIDATE

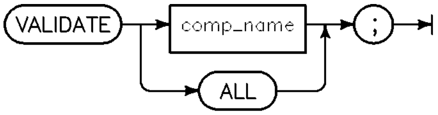
The VALIDATE statement is an action statement. This statement is one of three that control the state of a component. The other two are the DONTCARE and INVALIDATE statements.

VALIDATE is used to indicate to the development environment that the value of a component reflects the configuration of the instrument. This way, when the development environment recalls a state, it will know that this component does not need to be SET again.

You can use VALIDATE to indicate that a component is no longer DONTCARE, but the value in the component must match that of the instrument.

Note that a component is validated when it is referenced in an OUTPUT, BITS, or STORE statement, unless its state is DONTCARE.

Syntax



Item	Description
comp_name	The name of the component that you are specifying to be VALID.
ALL	You can use this keyword to mark all components in the HP ID as INVALID.

8

Example

```
REVISION 2.0;

COMPONENT ARANGE;
    TYPE DISCRETE;
    VALUES OFF,ON;
    INITIAL ON;
    SET ACTIONS;
        SELECT ARANGE;
        CASE OFF;
            OUTPUT STRING "AERH";
            INVALIDATE RANGE;
        CASE ON;
            OUTPUT STRING "AERA";
            DONTCARE RANGE;
        END SELECT;
    END ACTIONS;
END COMPONENT;

COMPONENT RANGE;
    TYPE INTEGER;
    VALUES RANGE 1,5;
    INITIAL DONTCARE;
    SET ACTIONS;
        OUTPUT RANGE FORMAT '"AERM",D,"EN"';
        FETCH (ARANGE)OFF;
        STORE ARANGE;
        VALIDATE RANGE;
    END ACTIONS;
END COMPONENT;
```

(Continued)

```
PANEL MAIN;
  POSITION 1,1;
  SIZE 200,200;
  DISCRETE ARANGE;
    POSITION 90,20;
    TITLE "A Range";
  END DISCRETE;
  CONTINUOUS RANGE;
    POSITION 90,50;
    TITLE "Range";
  END CONTINUOUS;
END PANEL;
```

Cross Reference

- DONTCARE
- INVALIDATE

VALUES

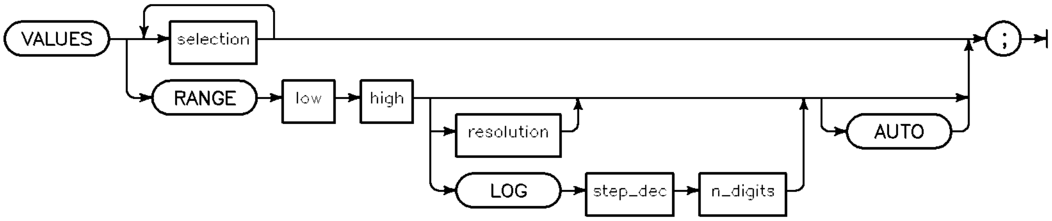
The VALUES statement is an optional part of all component descriptions except DISCRETE. It is required for DISCRETE components.

For numeric components this statement defines the RANGE of values that should be accepted by the development environment human interface. This may include the keyword AUTO for CONTINUOUS components.

The default VALUES for integers is -32768 to 32767. The default range for CONTINUOUS components is -1E18 through +1E18.

For a DISCRETE component, the VALUES statement enumerates every possible value for that component.

Syntax



VALUES

Item	Description
selection	This parameter is used only for DISCRETE components. The list of selections is the list of allowed values for DISCRETE components.
RANGE	This is used for all types of components other than DISCRETE and STRING. When using RANGE, you must specify the lowest value, highest value, and optional step size. The step size can be linear or logarithmic. The step size is used when the user changes the value of the component by clicking on the up/down arrows on the scroll bar rather than entering a value in the development environment.
low	A component name or number specifying the lowest value the user can enter for the component in the development environment.
high	A component name or number specifying the highest value the user can enter for the component in the development environment.
resolution	The amount the value is incremented or decremented when the user clicks on the arrows on the scroll bar in the development environment.
LOG	Specifies that the step size is logarithmic.
step_dec	The number of steps per decade.
n_digits	The number of digits. To increase by decade, (for example, 3, 30, 300): set **step_dec** = 1 and **n_digits** = 1 To increase in a 1-3 sequence: set **step_dec** = 2 and **n_digits** = 1 To increase in a 1-2-5 sequence: set **step_dec** = 3 and **n_digits** = 1
AUTO	For CONTINUOUS components only. Specifies that the control can have the value AUTO.

Example

```
REVISION 2.0;

COMPONENT RNG;
    TYPE CONTINUOUS;
    VALUES RANGE .001 100 LOG 1 1 AUTO;
END COMPONENT;

PANEL MAIN;
    SIZE 200,200;
    POSITION 1,1;

    CONTINUOUS RNG;
        POSITION 70,10;
        TITLE "Range";
    END CONTINUOUS;
END PANEL;
```

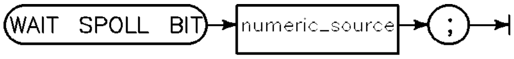
Cross Reference

- COMPONENT

WAIT SPOLL BIT

The WAIT SPOLL BIT statement is an action statement. This statement causes the development environment to suspend execution of the action list until it receives a serial poll bit (for example, ready bit) from the instrument.

Syntax



Item	Description
numeric_source	Specifies the bit number with 0 being the least significant bit. For complete information see “numeric_source” earlier in this chapter.

Example

```
REVISION 2.0;

COMPONENT READINGS;
    TYPE IARRAY 1 1024;
    GET ACTIONS;
        WAIT SPOLL BIT 4; ! WAIT UNTIL READY
        ENTER READINGS FORMAT INT16 0 1 512;
    END ACTIONS;
END COMPONENT;

PANEL MAIN;
    SIZE 200,200;
    POSITION 1,1;
END PANEL;
```

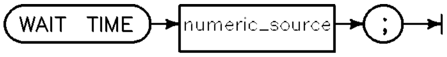
Cross Reference

- WAIT TIME

WAIT TIME

The WAIT TIME statement is an action statement. This statement causes the development environment to wait the specified number of seconds before executing the next statement.

Syntax



Item	Description
numeric_source	Specifies the number of seconds. For complete information see "numeric_source" earlier in this chapter.

Cross Reference

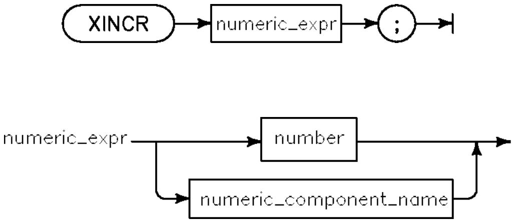
- WAIT SPOLL BIT

XINCR

The XINCR statement allows you to specify the X-axis increment between data points in an ITRACE or RTRACE if XLOG is 0. If XLOG is 1, it specifies the X-axis multiplier between points.

The default value is 1. The HP ID writer is responsible for updating this field. The field is transferred as part of the waveform record in an hpt_peek_waveform operation.

Syntax



Item	Description
numeric_expr	Specifies the X-axis increment between points. For linear traces this is the spacing between points. For log traces this is the multiplier between points.

Example

```
COMPONENT TRACE1;
  TYPE RTRACE 1024;
  TRACETYPE WAVEFORM;
  POINTS WF_SIZE;
  XMIN X_MIN;
  XINCR X_INCR;
  XLOG OFF;
  XUNIT "s";
  YUNIT "V";
```

(Continued)

```
COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;

COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- COMPONENT

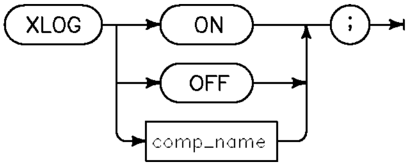
XLOG

The XLOG statement allows you to specify the X-axis spacing in an ITRACE or RTRACE.

XLOG is an integer. A value of 0 (OFF) specifies linear spacing. A non-0 (ON) value specifies log spacing. The default value is OFF.

The HP ID writer is responsible for updating this field. The field is transferred as part of the waveform record in an hpt_peek_waveform operation.

Syntax



Item	Description
numeric_expr	0=linear X-axis spacing. Non 0=logarithmic X-axis spacing.

Example

```
TYPE RTRACE 1024;
TRACETYPE WAVEFORM;
POINTS WF_SIZE;
XMIN X_MIN;
XINCR X_INCR;
XLOG OFF;
XUNIT "s";
YUNIT "V";
```

(Continued)

```
COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;

COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- COMPONENT

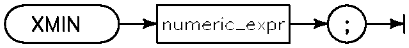
XMIN

The XMIN statement allows you to specify the X-axis value for the initial data point in an ITRACE or RTRACE.

The default value is 0. The HP ID writer is responsible for updating this field. The field is transferred as part of the waveform record in an hpt_peek_waveform operation.

Item	Description
numeric_expr	Specifies the value for the initial data point of a waveform.

Syntax



Example

```
COMPONENT TRACE1;
  TYPE RTRACE 1024;
  TRACETYPE WAVEFORM;
  POINTS WF_SIZE;
  XMIN X_MIN;
  XINCR X_INCR;
  XLOG OFF;
  XUNIT "s";
  YUNIT "V";
```

(Continued)

```
COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;

COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

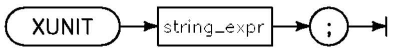
- COMPONENT

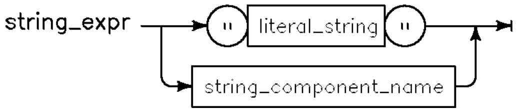
XUNIT

The XUNIT statement allows you to specify the unit for the X axis in an ITRACE or RTRACE.

The statement is of type string and the default value is the null string. The HP ID writer is responsible for updating this field. The field is transferred as part of the waveform record in an hpt_peek_waveform operation.

Syntax





Item	Description
string_expr	Specifies a string used by the measurement product for graph labelling of the X axis.

Example

```
COMPONENT TRACE1;
  TYPE RTRACE 1024;
  TRACETYPE WAVEFORM;
  POINTS WF_SIZE;
  XMIN X_MIN;
  XINCR X_INCR;
  XLOG OFF;
  XUNIT "s";
  YUNIT "V";
```

(Continued)

```
COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;

COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

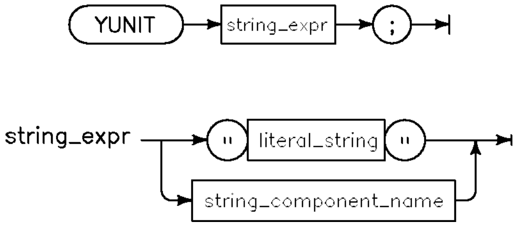
- COMPONENT

YUNIT

The YUNIT statement allows you to specify the unit for the Y axis in an ITRACE or RTRACE.

The statement is of type string and the default value is the null string. The HP ID writer is responsible for updating this field. The field is transferred as part of the waveform record in an hpt_peek_waveform operation.

Syntax



Item	Description
string_expr	Specifies a string used by the measurement product for graph labelling of the Y axis.

Example

```
COMPONENT TRACE1;
  TYPE RTRACE 1024;
  TRACETYPE WAVEFORM;
  POINTS WF_SIZE;
  XMIN X_MIN;
  XINCR X_INCR;
  XLOG OFF;
  XUNIT "s";
  YUNIT "V";
```

(Continued)

```
COMPONENT X_INCR;
  TYPE CONTINUOUS;
  INITIAL 2E-6;
  GET ACTIONS;
    FETCH TIME_SENS; FETCH 10; MUL;
    FETCH WF_SIZE; DIV;
    STORE X_INCR;
  END ACTIONS;
END COMPONENT;

COMPONENT X_MIN;
  TYPE CONTINUOUS;
  INITIAL -500E-6;
  GET ACTIONS;
    SELECT TIME_REF;
    CASE LEFT;
      FETCH 0;
    CASE CENTER;
      FETCH TIME_SENS; FETCH -5; MUL;
    CASE RIGHT;
      FETCH TIME_SENS; FETCH -10; MUL;
    END SELECT;
    FETCH TIME_DELAY; ADD;
    STORE X_MIN;
  END ACTIONS;
END COMPONENT;
```

Cross Reference

- COMPONENT

9

Panel Syntax

Overview

This section contains an alphabetical listing of the keywords currently available for use in the panel section of an HP ID. Each entry defines the keyword, shows the proper syntax for its use, provides one or more examples, and explains relevant semantic details.

Interpreting the Syntax Drawings

- All characters enclosed by an **oval** must be entered exactly as shown (case-independent).
- Words enclosed by a **rectangle** are names of items used in the statement.
- *Italic* letters indicate that the word or words are keywords, and are fully documented elsewhere in this chapter.
- Statement elements are connected by lines. Each line can be followed in only one direction with an arrow indicating the direction.

Optional Elements and Defaults

An element is optional if there is a path around it. Optional values usually have default values. The table or text following the drawing specifies the default value that is used when an optional item is not included in a statement.

Naming Rules

- Don't use keywords for component or action names.
- Component and action names can be up to 25 characters long.
- Component and action names must start with an alpha character, A-Z or a-z. That character may be followed by any mix of alphanumeric characters, either upper or lower case, or by an underscore. The syntax is not case sensitive.

  Following is a list of possible component names:

  - FREQUENCY
  - SLOT4
  - SLOT_4
  - Start_Frequency

Comments

A comment may be created by preceding a statement with an exclamation mark. You can also make comments on the same line as a statement. This is done by placing an exclamation mark after the statement.

Spaces, Commas, and Other Separators

In general, a space is required between a keyword and an item. A space or a comma is required between multiple items following a keyword. Tabs are treated as spaces.

BACKGROUND

BACKGROUND specifies the area fill color of a panel or panel element. For color displays, BACKGROUND specifies intensities of red, green, and blue. For monochrome displays, BACKGROUND specifies the percentage of white.

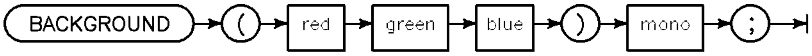
On a monochrome display, the development environment ignores the values provided for red, green, and blue. On a color display, it ignores the values provided for mono.

The development environment supports up to 16 colors (see table 9-1). When the user selects a panel's control or display in the development environment, it highlights the selected control or display in a color that is the inverse of the original color.

Note

The actual number of supported colors depends on the particular computer system you use.

Syntax



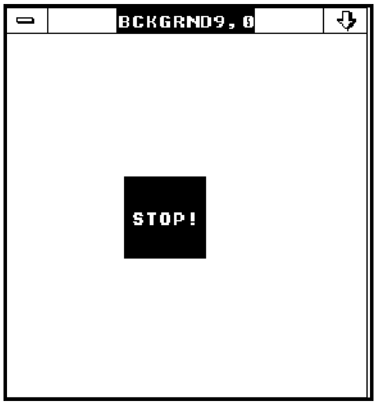
Item	Description	Range
red	An integer	0-255
green	An integer	0-255
blue	An integer	0-255
mono	An integer	0=black 100=white

Table 9-1. Recommended Colors for Panel Elements

Color Name	Example	Red	Green	Blue
Lavender	Title Bar	90	0	170
Evening Blue	System Menu Bar	75	0	240
Beige Gray	Background	148	139	123
French Gray	Editor Background	105	95	80
Evening Gold	Pushbuttons	150	110	75
Forest Green	Reset Buttons	0	130	70
Parchment White	Panel Background	220	211	184
White	Panel Outline	255	255	255
Black	Panel Text	0	0	0
Yellow	1st Trace on XY	255	255	0
Cyan	2nd Trace on XY	0	255	255
Magenta	3rd Trace on XY	255	0	255
Rich Green	4th Trace on XY	0	200	50
Harvest Gold	None	238	150	0
Safety Red	None	230	30	30
Razor Blue	None	120	184	210

Example

```
REVISION 2.0;
COMPONENT Btn;
    TYPE INTEGER;
    SET ACTIONS;
        NOTIFY "Ok";
    END ACTIONS;
END COMPONENT;
PANEL Main;
    BUTTON Btn;
        POSITION 70,80;
        SIZE 50,50;
        LABEL "STOP!";
        BACKGROUND (230,30,30),100;
        FOREGROUND (255,255,255),0;
    END BUTTON;
END PANEL;
```



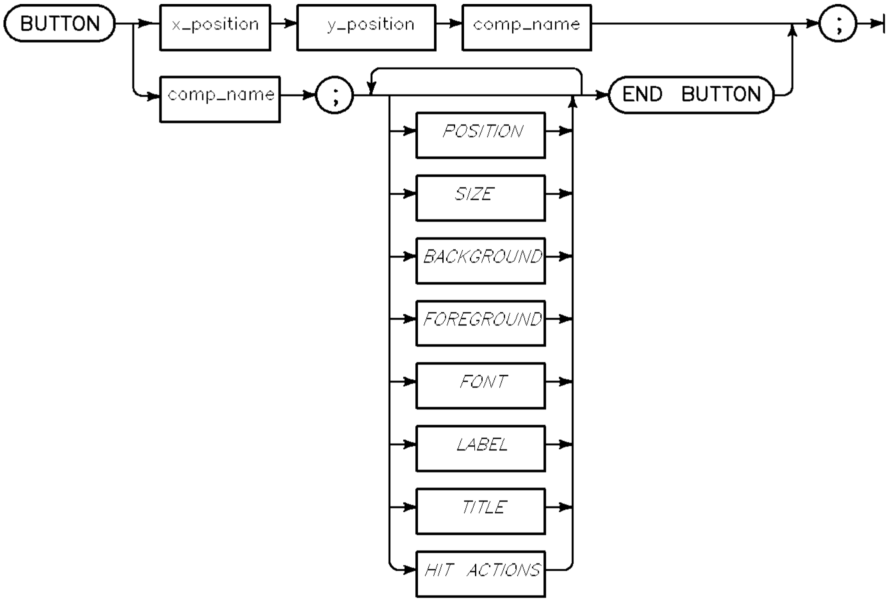
Cross Reference

- BUTTON
- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- PANEL
- TEXT
- TOGGLE
- XY

BUTTON

The BUTTON ... END BUTTON compound statement allows you to simulate a momentary contact button. When the user clicks on a button in the development environment, it immediately executes the SET ACTIONS specified in the component description associated with the button. The component that the BUTTON is tied to is set to a value of 1.

Syntax



BUTTON

Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the button.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the button.
comp_name	The name of an INTEGER component whose SET ACTIONS are executed when the button is pressed (see "HIT ACTIONS" in this chapter).
POSITION	Consists of an x-position and a y-position that together specify the lower left corner of the button. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels for the characters displayed in LABEL. Default is 9 x 15 pixels.

SIZE	Consists of a width and height that together specify the size in pixels of the button. If not specified, the development environment computes a default size using the following formula:
	width = (length of LABEL × FONT width) + 4
	height = FONT height + 4

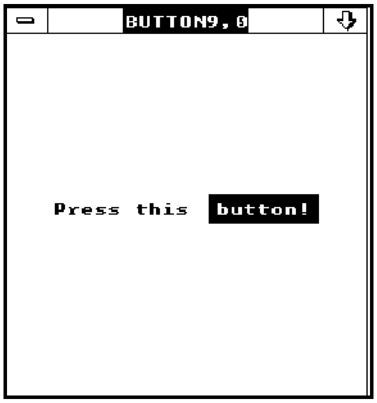
BACKGROUND	Specifies the area fill color for the button. Values must be provided for red, green, blue, and monochrome. Default color is Evening Gold (see "BACKGROUND," table 9-1) and black for monochrome displays.
FOREGROUND	Specifies the color of the text and border of the button. Values must be provided for red, green, blue, and monochrome. Default color is White for color (see table 9-1) and monochrome displays.
LABEL	Specifies the text to be placed inside of the button. If more than one string is specified, the development environment uses only the first. Default is the associated component name.

(Continued)

Item	Description
HIT ACTIONS	If specified, the development environment executes this action list instead of the SET ACTIONS of the associated component. Executed only in the development environment.
TITLE	Specifies the text to be placed to the left of the button on the panel. If not specified, the development environment does not generate one. The color of the characters in the TITLE is black.

Example

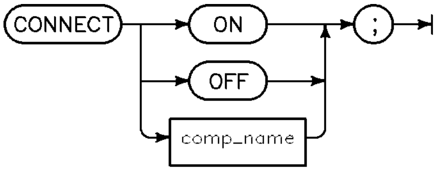
```
REVISION 2.0;
COMPONENT Btn;
    TYPE INTEGER;
    SET ACTIONS;
        NOTIFY "The button was pressed.";
    END ACTIONS;
END COMPONENT;
PANEL Main;
    BUTTON Btn;
        POSITION 120,100;
        TITLE "Press this ";
        LABEL "button!";
    END BUTTON;
END PANEL;
```



CONNECT

CONNECT is used to turn on and off the connect-the-dots mode for a TRACE in the XY panel element.

Syntax



Item	Description
ON	Specifies that the dots should be connected.
OFF	Specifies that the dots should not be connected.
comp_name	The name of an INTEGER, CONTINUOUS, or DISCRETE component.

For INTEGER or CONTINUOUS components:

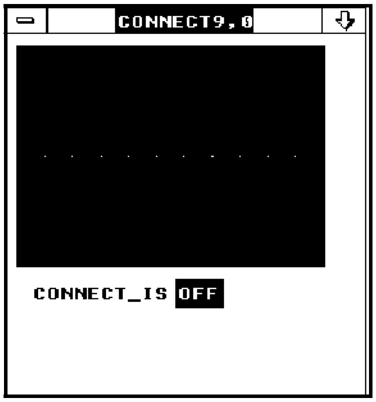
- If the value is 0, then CONNECT is off. Otherwise it's on.

For DISCRETE components:

- If the current value is the first in the list, then CONNECT is off. Otherwise it's on.

Example

```
REVISION 2.0;

COMPONENT Trace_data;
    TYPE RARRAY 1,10;
END COMPONENT;

COMPONENT Connect_is;
    TYPE DISCRETE;
    VALUES OFF,ON;
END COMPONENT;

PANEL Main;
    XY;
        TRACE Trace_date;
            SCALE 0,11,-5,5;
            CONNECT Connect_is;
        END TRACE;
    END XY;

        TOGGLE 100,50,Connect_is;
END PANEL;
```



Cross Reference

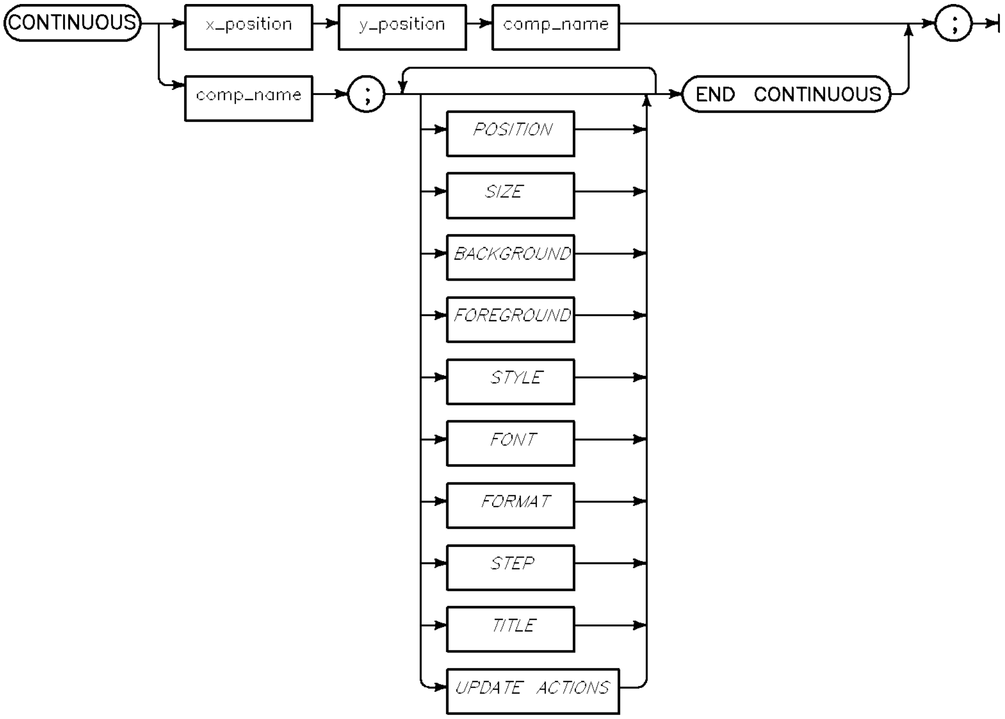
- TRACE

CONTINUOUS

The CONTINUOUS ... END CONTINUOUS compound statement creates a control that allows the user to input a numeric value within a specified range. You can specify the range, including AUTO if appropriate, using the RANGE option in the VALUES statement in the component description.

The current value of a CONTINUOUS control is displayed in a box on the panel; the development environment displays ? in the box for INVALID values, * for components specified as DONTCARE, and AUTO if auto is enabled.

Syntax



Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the control.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the control.
comp_name	The name of an INTEGER or CONTINUOUS component whose SET ACTIONS are executed after the user has entered a new value. The value displayed is stored in this component.
POSITION	Consists of an x-position and a y-position that together specify the lower left corner of the control. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels of the characters associated with this control. These include the characters that constitute the value and the TITLE. Default is 9 x 15 pixels.
FORMAT	Requires a string in which you insert either specifiers (table 9-4 under FORMAT keyword) or a number and the word DIGITS. The latter indicates that you want the current value rounded to the specified number of digits. Default is 3 DIGITS (for example, 4.33).
STYLE	Requires a string. If the string is "NOENGR", then the development environment does not display engineering suffixes (see table 9-2). Default provides engineering suffixes. The development environment determines the appropriate suffixes based on the range specified in the associated component. NOENGR = no engineering suffixes used anything else in the string = engineering suffixes used NONDECIMAL = no suffixes are displayed. Instead, the entry box contains additional fields for binary, octal, decimal, and hexadecimal entry. The display format is controlled by the FORMAT specifier.

(Continued)

Item	Description

SIZE	Consists of a width and height that together specify the size in pixels of the control. If not specified, the development environment computes a default size using the following formula:
	If STYLE "NOENGR" is not specified:
	width = ((length of FORMAT + 3) × FONT width) + 4
	height = FONT height + 4
	If STYLE "NOENGR" is specified:
	width = (length of FORMAT × FONT width) + 4
	height = FONT height + 4

BACKGROUND	Specifies the area fill color for the control. Values must be provided for red, green, blue, and monochrome. Default color is French Gray (see "BACKGROUND," table 9-1) and black for monochrome displays.
FOREGROUND	Specifies the color of the text and border of the control. Values must be provided for red, green, blue, and monochrome. Default color is White (see table 9-1) for color and monochrome displays.
STEP	Specifies the amount by which the up/down arrows on the scroll bar change the current value. If STEP is specified, then resolution is continuous (that is, any number between the upper and lower limits is acceptable). You can specify STEP size to be linear or logarithmic. If STEP is not specified, then the up/down arrows change the value by the resolution specified in the component (see "VALUES" under "RANGE" in chapter 8, "Component/Action Syntax").
TITLE	Specifies the text to be placed to the left of the control. If you do not provide a TITLE, the development environment does not generate one. However, if you use the short form of the CONTINUOUS statement (that is, specify just the x_position, y_position, and comp_name), then the development environment generates a default TITLE, which is comp_name.
UPDATE ACTIONS	An action list whose statements the development environment executes whenever the view of the associated component is changed in the development environment.

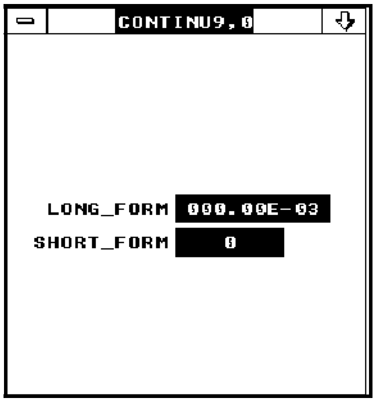
Table 9-2. Engineering Suffixes Available

Symbol	Prefix	Multiplication Factor
E	exa	10^{18}
P	peta	10^{15}
T	tera	10^{12}
G	giga	10^{9}
M	mega	10^{6}
k	kilo	10^{3}
m	milli	10^{-3}
u	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Example

```
REVISION 2.0;

COMPONENT Short_form;
    TYPE CONTINUOUS;
END COMPONENT;

PANEL Main;
    CONTINUOUS Short_form;
        POSITION 100,100;
        FORMAT "DDD.DDESZZ";
        STYLE "NOENGR";
        TITLE "LONG_FORM";
    END CONTINUOUS;

    CONTINUOUS 100,80,Short_form;
END PANEL;
```

CONTINUOUS



```
LONG_FORM  000.00E-03
SHORT_FORM     0
```

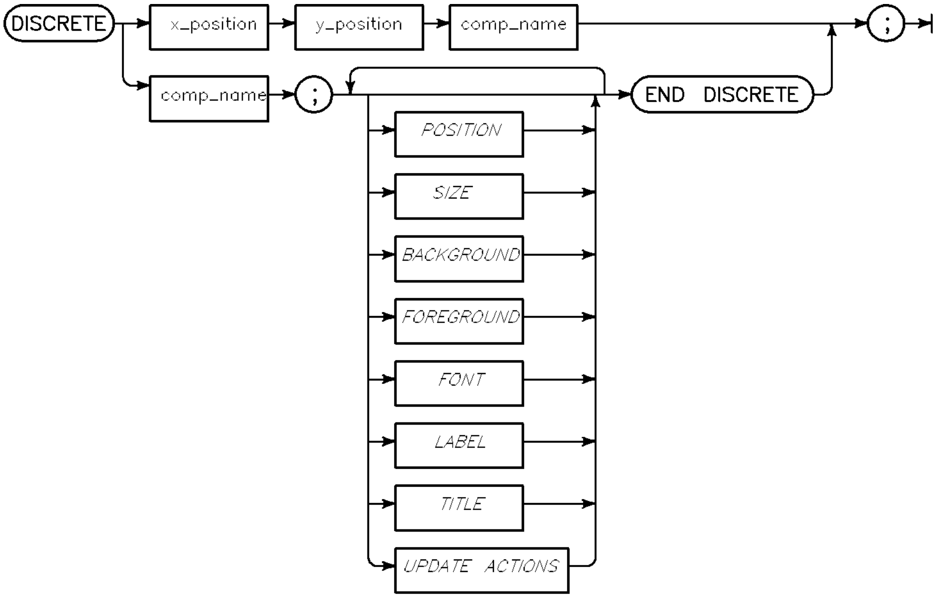
DISCRETE

The DISCRETE ... END DISCRETE compound statement allows you to create a control in which the user can select from a list of allowable values.

The current value of a DISCRETE control is displayed in a box on the panel; the development environment displays ? in the box for INVALID values and * for components marked as DONTCARE. When the user clicks on the box containing the current value, the development environment displays the list of valid values from which the user can select a new value.

A DISCRETE control and a CONTINUOUS control look identical on a panel. The only difference is the way in which the user changes the current value.

Syntax



DISCRETE

Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the control.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the control.
comp_name	The name of a DISCRETE component whose SET ACTIONS are executed after the user has selected a new value. The value displayed is stored in this component.
POSITION	Consists of an x-position and a y-position that together specify the lower left corner of the control. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels of the characters associated with this control. These include the characters that constitute the value and the TITLE. Default is 9 x 15 pixels.

SIZE	Consists of a width and height that together specify the size in pixels of the control. If not specified, the development environment computes a default size using the following formula:
	width = (length of longest LABEL × FONT width) + 4
	height = FONT height + 4

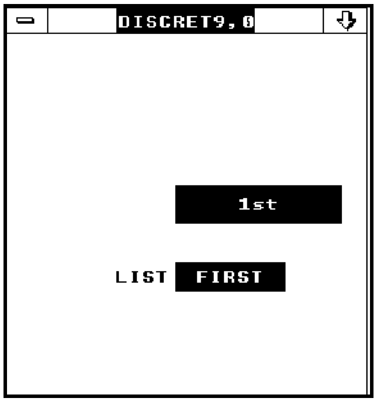
BACKGROUND	Specifies the area fill color for the control. Values must be provided for red, green, blue, and monochrome. Default color is French Gray (see "BACKGROUND," table 9-1) and black for monochrome displays.
FOREGROUND	Specifies the color of the text and border of the control. Values must be provided for red, green, blue, and monochrome. Default color is White (see table 9-1) for color and monochrome displays.

(Continued)

Item	Description
LABEL	Specifies the selections listed in the list box that the development environment displays when the user clicks on the control box. The first string in the LABEL statement corresponds to the first VALUE specified in the associated component, and so on. The development environment ignores any excess strings. If LABEL is not specified, the development environment uses the VALUES statement in the component as the default labels. If there are more VALUES than LABEL strings, then the development environment uses the excess VALUES as the default labels.
TITLE	Specifies the text to be placed to the left of the control. If you do not provide a TITLE, the development environment does not generate one. The color of the characters in the TITLE is black. However, if you use the short form of the DISCRETE statement (that is, specify just the x_position, y_position, and comp_name), then the development environment generates a default TITLE, which is comp_name.
UPDATE ACTIONS	An action list whose statements the development environment executes whenever the view of the associated component is changed in the development environment.

Example

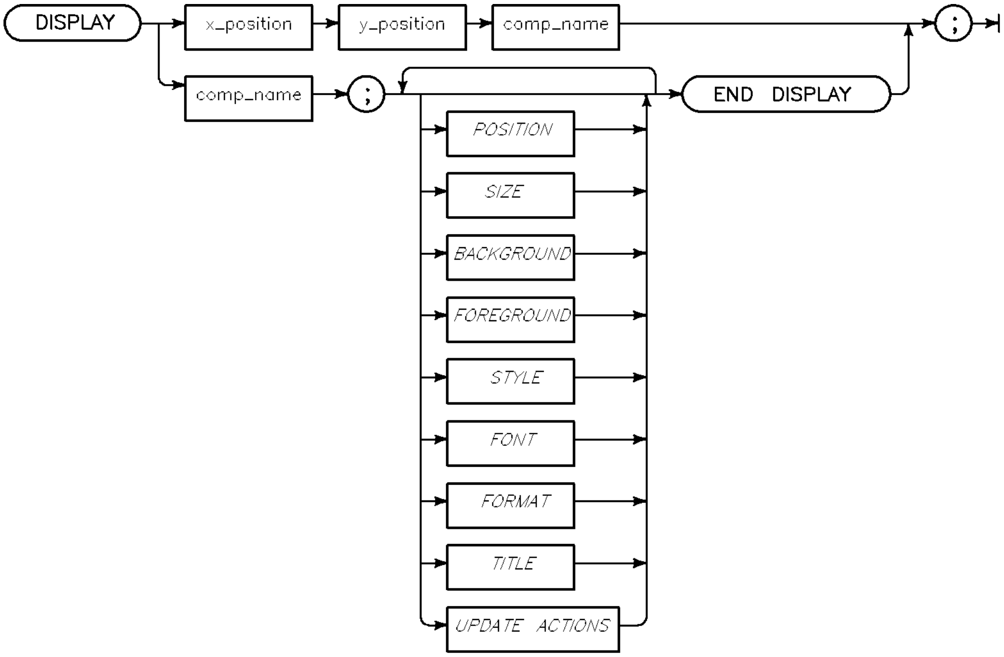
```
REVISION 2.0;

COMPONENT List;
    TYPE DISCRETE;
    VALUES First,Second,Third,Fourth,Fifth,Sixth,Seventh;
END COMPONENT;

PANEL Main;
    DISCRETE List;
        POSITION 100,100;
        SIZE 100,25;
        LABEL "1st","2nd","3rd","4th","5th","6th","7th";
    END DISCRETE;

    DISCRETE 100,60,List;
END PANEL;
```



DISPLAY

The DISPLAY ... END DISPLAY compound statement allows you to create an area in which the development environment can display an instrument reading or measurement.

Syntax



DISPLAY

Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the display.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the display.
comp_name	The name of a CONTINUOUS, INTEGER, DISCRETE, or STRING component whose GET ACTIONS are executed whenever the user clicks on the display (see "HIT ACTIONS" in this chapter).
POSITION	Consists of an x_position and a y_position that together specify the lower left corner of the display. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels of the characters in the display. Default is 9 x 15 pixels.
STYLE	Only valid for DISPLAYs associated with INTEGER or CONTINUOUS components. Requires a string. If the string is NOENGR, then the development environment does not display any engineering suffixes with numeric values. Default provides engineering suffixes (see "CONTINUOUS," table 9-2). NOENGR = no engineering suffixes used. anything else in the string = engineering suffixes used.
FORMAT	Requires a string in which you insert either specifiers (FORMAT, table 9-4) or a number and the word DIGITS. The latter indicates that you want the current value rounded to the specified number of digits. Default is 3 DIGITS (for example, 4.33).

(Continued)

Item	Description

SIZE	Consists of a width and height that together specify the size in pixels of the display. If not specified, the development environment computes a default size using the following formula:
	If the component is an INTEGER or CONTINUOUS:
	If STYLE "NOENGR" is not specified:
	width = ((length of FORMAT + 3) × FONT width) + 4
	height = FONT height + 4
	If STYLE "NOENGR" is specified:
	width = (length of FORMAT × FONT width) + 4
	height = FONT height + 4
	If the component is a DISCRETE:
	width = (length of longest LABEL × FONT width) + 4
	height = FONT height + 4
	If the component is a STRING:
	width = (length of FORMAT × FONT width) + 4
	height = FONT height + 4

BACKGROUND	Specifies the area fill color for the display. Values must be provided for red, green, blue, and monochrome. Default color is French Gray (see "BACKGROUND," table 9-1), and black for monochrome displays.
FOREGROUND	Specifies the color of the text and border of the display. Values must be provided for red, green, blue, and monochrome. Default color is White (see table 9-1) for both color and monochrome displays.

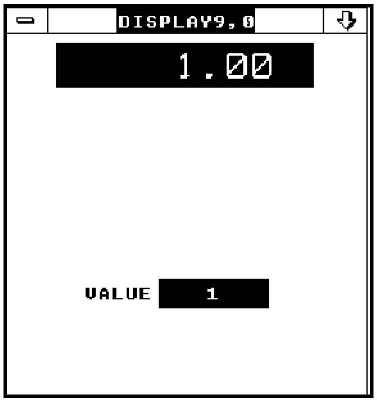
(Continued)

DISPLAY

Item	Description
TITLE	Specifies the text to be placed to the left of the display on the panel. If not specified, the development environment does not generate a title. The color of the characters in the TITLE is black.
UPDATE ACTIONS	An action list whose statements the development environment executes whenever the view of the associated component is changed in the development environment.

Example

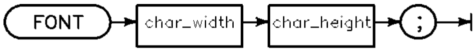
```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
    END ACTIONS;
END COMPONENT;                    ! Reset

COMPONENT Value;
    TYPE CONTINUOUS;
    VALUES RANGE -1000,1E6,.001;
    INITIAL 1;
END COMPONENT;                    !Value

PANEL Main;
    DISPLAY Value;
        POSITION 30,180;
        FONT 15,25;
        FORMAT "DDDD.DD";
    END DISPLAY;

    CONTINUOUS 90,50,Value;
END PANEL;                        ! Main
```



FONT

FONT consists of a width and a height that together specify the size in pixels of each character in the panel elements that display text. The default is 9 × 15 pixels. For consistency, Hewlett-Packard recommends that you only use two font sizes, a 9 × 15 font size for most text and a 15 × 25 font size for the text in large DISPLAYs.

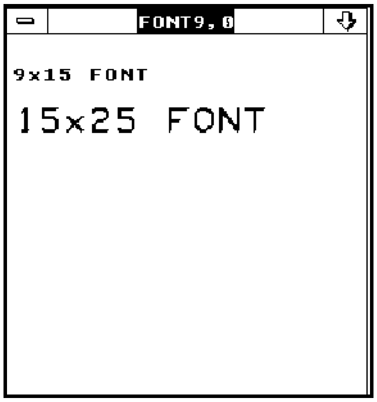
Syntax



Item	Description	Range
char_width	An integer	1 - 32767
char_height	An integer	1 - 32767

Example

```
REVISION 2.0;

PANEL Main;

    TEXT 5,180,"9x15 FONT";

    TEXT "15x25 FONT";
        POSITION 5,150;
        FONT 15,25;
    END TEXT;
END PANEL;
```



9x15 FONT

15x25 FONT

Cross Reference

- BUTTON
- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- TEXT
- TOGGLE

FOREGROUND

FOREGROUND specifies the line and text color of a panel, subpanel, or panel element. For color displays, FOREGROUND specifies intensities of red, green, and blue. For monochrome displays, FOREGROUND specifies the percentage of white.

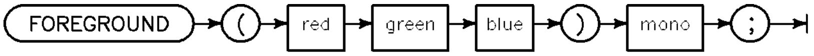
On a monochrome display, the development environment ignores the values provided for red, green, and blue. On a color display, it ignores the values provided for mono.

The development environment supports up to 16 colors (see table 9-3). When the user selects a panel's control or display in the development environment, it highlights the selected control or display in a color that is the inverse of the original color.

Note

The actual number of supported colors depends on the particular computer system you use.

Syntax



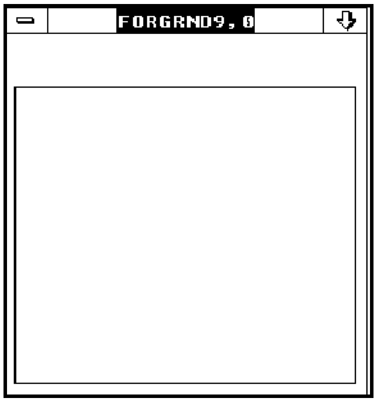
Item	Description	Range
red	An integer	0-255
green	An integer	0-255
blue	An integer	0-255
mono	An integer	0=black 100=white

Table 9-3. Recommended Colors for Panel Elements

Color Name	Example	Red	Green	Blue
Lavender	Title Bar	90	0	170
Evening Blue	System Menu Bar	75	0	240
Beige Gray	Background	148	139	123
French Gray	Editor Background	105	95	80
Evening Gold	Pushbuttons	150	110	75
Forest Green	Reset Buttons	0	130	70
Parchment White	Panel Background	220	211	184
White	Panel Outline	255	255	255
Black	Panel Text	0	0	0
Yellow	1st Trace on XY	255	255	0
Cyan	2nd Trace on XY	0	255	255
Magenta	3rd Trace on XY	255	0	255
Rich Green	4th Trace on XY	0	200	50
Harvest Gold	None	238	150	0
Safety Red	None	230	30	30
Razor Blue	None	120	184	210

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        SHOW Sub;
    END ACTIONS;
END COMPONENT;

PANEL Main;
    PANEL Sub;
        FOREGROUND (230,30,30),100;
    END PANEL;
END PANEL;
```



Cross Reference

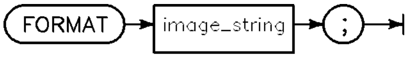
- BUTTON
- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- MARKER
- PANEL
- TEXT
- TOGGLE
- TRACE
- XY

FORMAT

FORMAT allows you to specify how numbers are displayed. FORMAT requires a string in which you insert either specifiers (see table 9-4) or an integer followed by the word DIGITS. The latter indicates that you want the current value rounded to the specified number of digits.

In general, the default FORMAT is “3 DIGITS” (for example, 4.33).

Syntax



Item	Description
image_string	Table 9-4 lists the valid specifiers. You can also specify an integer followed by the word DIGITS. The following applies to DIGITS: ■ If the integer is >15, no rounding occurs. ■ If the integer is <1, 0 is returned. ■ If the integer is 1 - 15, the value is rounded off to that many digits.

Table 9-4. Valid FORMAT Specifiers

Specifier	Meaning
K	Compact field. Prints the number in standard form with no leading or trailing blanks.
-K	Same as K.
H	Similar to K, except that the number is printed using the European number format (comma radix).
-H	Same as H.
S	Outputs the number's sign (+ or -).
M	Prints the number's sign if negative, a blank if positive.
D	Prints 1-digit character. A leading zero is replaced by a blank. If the number is negative and no sign is specified, the minus sign will occupy a leading digit position. If a sign is printed, it will "float" to the left of the left-most digit.
Z	Same as D, except that leading zeros are printed.
*	Like Z, except that asterisks are printed instead of leading zeros.
.	Prints a decimal-point radix indicator.
R	Prints a comma radix indicator (European format).
E	Prints an E, a sign, and a two-digit exponent.
ESZ	Prints an E, a sign, and a one-digit exponent.
ESZZ	Same as E.
ESZZZ	Prints an E, a sign, and a three-digit exponent.
A	Outputs a string character.
literal	Outputs the characters in the literal.
B	Output a byte. The number is rounded to an INTEGER and the least significant byte is output.
#	Statement is terminated when the last ENTER item is terminated. EOI and line feed are item terminators and early termination is not allowed.
X	Skips a character.

Table 9-5. Valid FORMAT Specifiers (cont.)

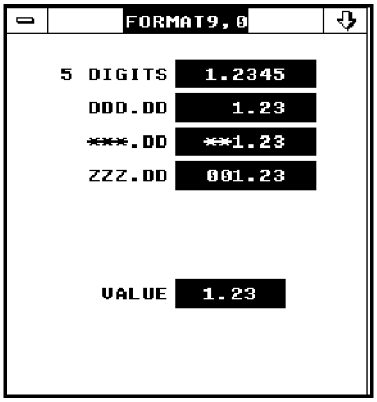
Specifier	Meaning
h	Prints 1 hexadecimal character. Leading 0s are printed. Signs are not printed. Instead, they are replaced by 2's complement representation of the hex number.
o	Same as h, except that an octal digit is printed.
b	Same as h, except that a binary digit is printed.

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
    END ACTIONS;
END COMPONENT;

COMPONENT Value;
    TYPE CONTINUOUS;
    INITIAL 1.2345;
END COMPONENT;

PANEL Main;
    DISPLAY Value;
        POSITION 100,180;
        FORMAT "5 DIGITS";
        TITLE "5 DIGITS";
    END DISPLAY;

    DISPLAY Value;
        POSITION 100,160;
        FORMAT "DDD.DD";
        TITLE "DDD.DD";
    END DISPLAY;
```

(Continued)

```
    DISPLAY Value;
        POSITION 100,140;
        FORMAT "***.DD";
        TITLE "***.DD";
    END DISPLAY;

    DISPLAY Value;
        POSITION 100,120;
        FORMAT "ZZZ.DD";
        TITLE "ZZZ.DD";
    END DISPLAY;

    CONTINUOUS 100,50,Value;
END PANEL;
```



5 DIGITS 1.2345
DDD.DD 1.23
***.DD **1.23
ZZZ.DD 001.23

VALUE 1.23

Cross Reference

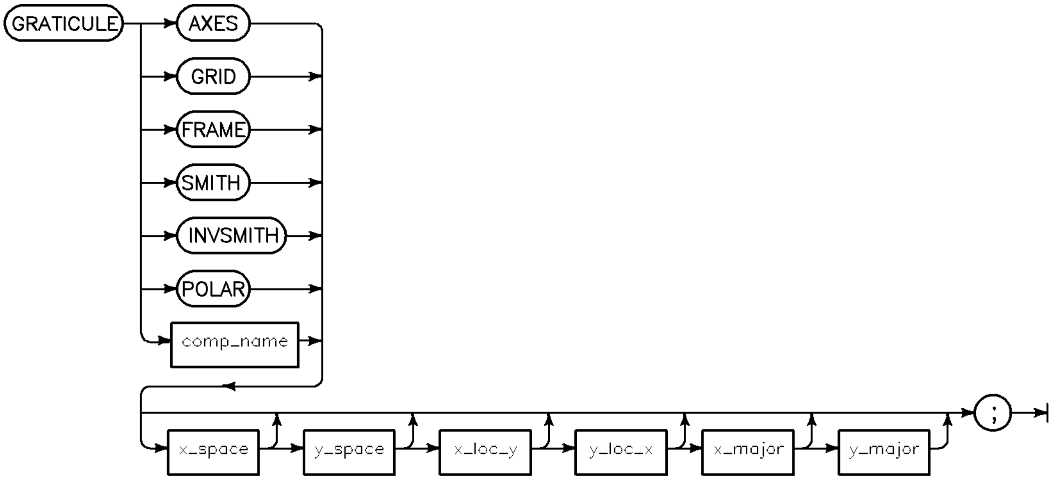
- CONTINUOUS
- DISPLAY
- INPUT

GRATICULE

GRATICULE specifies the graticule to be drawn on an XY display. The default is no GRATICULE.

Note Unless you specify SCALE, the development environment uses pixels to scale the graticule.

Syntax



Item	Description
AXES	Specifies that the development environment draw an XY display with an X axis and Y axis.
GRID	Specifies that the development environment draw an XY display with vertical and horizontal lines at the tic marks on the x and y axes.
FRAME	Specifies that the development environment draw an XY display with tic marks around the edge.
comp_name	Specifies a DISCRETE or INTEGER component whose value determines the type of graticule (see table 9-5).
x_space	Specifies the tic mark spacing along the X axis. Default is 1.
y_space	Specifies the tic mark spacing along the Y axis. Default is 1.
x_loc_y	Specifies the intersection points of the X axis along the orthogonal axis. Default is 0. This value is ignored if the type is a FRAME.
y_loc_x	Specifies the intersection points of the Y axis along the orthogonal axis. Default is 0. This value is ignored if the type is a FRAME.
x_major	Specifies the number of tick intervals between major tick marks along the X axis. Default is 1 (every tick is major).
y_major	Specifies the number of tick intervals between major tick marks along the Y axis. Default is 1 (every tick is major).
SMITH	Specifies that the development environment draw a Smith chart graticule. Smith chart scaling (reference value) is taken from the "x_space" component.
INVSMITH	Specifies that the development environment draw an Inverted Smith chart graticule. Inverted Smith chart scaling (reference value) is taken from the "x_space" component.
POLAR	Specifies that the development environment draws a polar graticule.

Note

For SMITH/POLAR graticules, the spacing, loc, and major information is not used if it is specified.

GRATICULE

Table 9-6. The Value of comp_name Controls the Graticule

Numeric Value	DISCRETE Position	Graticule Type
0	First	None
1	Second	AXES
2	Third	GRID
3	Fourth	FRAME
4	Fifth	SMITH
5	Sixth	INVSMITH
6	Seventh	POLAR

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
        VALIDATE Xydata;
    END ACTIONS;
END COMPONENT;

COMPONENT Xydata;
    TYPE RARRAY 5,1;
END COMPONENT;
```

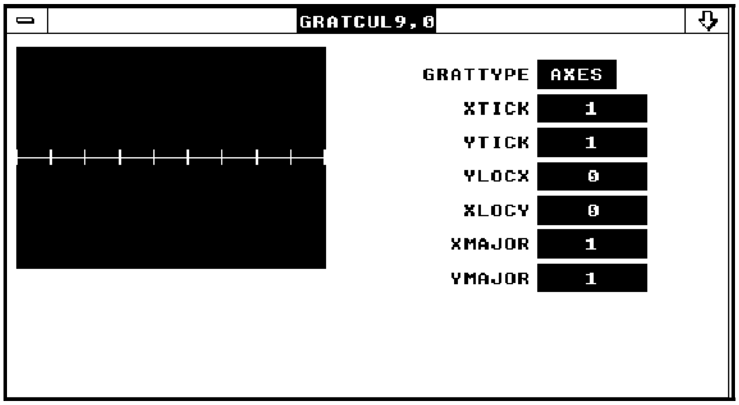
(Continued)

```
COMPONENT Grattype;
    TYPE DISCRETE;
    VALUES NONE,AXES,GRID,FRAME;
    INITIAL AXES;
END COMPONENT;

COMPONENT Xtick;
    TYPE REAL;
    INITIAL 1;
END COMPONENT;

COMPONENT Ytick;
    TYPE REAL;
    INITIAL 1;
END COMPONENT;

COMPONENT Xlocy;
    TYPE REAL;
    INITIAL 0;
END COMPONENT;

COMPONENT Ylocx;
    TYPE REAL;
    INITIAL 0;
END COMPONENT;

COMPONENT Xmajor;
    TYPE REAL;
    INITIAL 1;
END COMPONENT;
```

(Continued)

GRATICULE

```
COMPONENT Ymajor;
    TYPE REAL;
    INITIAL 1;
END COMPONENT;

PANEL Main;
    SIZE 428,213;
    XY;
      SCALE 1,10,-5,5;
      GRATICULE Grattype,Xtick,Ytick,Ylocx,Xlocy,Xmajor,Ymajor;
      TRACE Xydata;
      END TRACE;
    END XY;

    DISCRETE 314,180,Grattype;
    CONTINUOUS 314,160,Xtick;
    CONTINUOUS 314,140,Ytick;
    CONTINUOUS 314,120,Ylocx;
    CONTINUOUS 314,100,Xlocy;
    CONTINUOUS 314,80,Xmajor;
    CONTINUOUS 314,60,Ymajor;
END PANEL;
```



GRATTYPE	AXES
XTICK	1
YTICK	1
YLOCX	0
XLOCY	0
XMAJOR	1
YMAJOR	1

Cross Reference

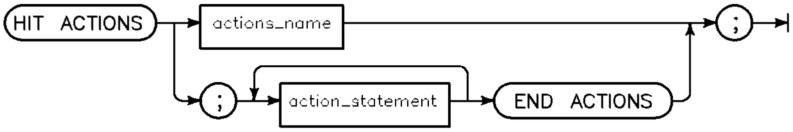
- SCALE
- XY

HIT ACTIONS

The development environment executes the action statements within this action list rather than the SET ACTIONS or GET ACTIONS for the associated component.

Caution If you use HIT ACTIONS, then you may not generate any code automatically for the panel element or even be able to access the component. This depends on the development product you are using.

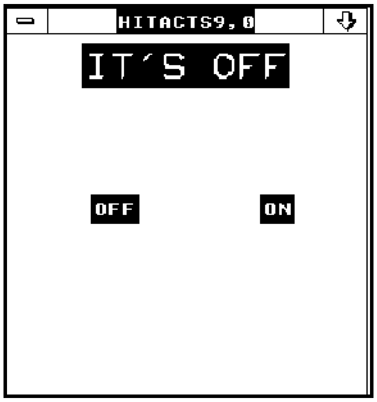
Syntax



Item	Description
action_statement	See tables 8-1 through 8-8 under "ACTIONS" in chapter 8, "Component/Action Syntax," for the valid statements.
actions_name	The name of the action list.

Example

```
REVISION 2.0;

COMPONENT Value;
    TYPE DISCRETE;
    VALUES OFF,ON;
END COMPONENT;

COMPONENT Btn_dummy;
    TYPE INTEGER;
END COMPONENT;

PANEL Main;
    DISCRETE Value;
        POSITION 45,180;
        FONT 15,25;
        LABEL "IT'S OFF","IT'S ON";
    END DISCRETE;

    BUTTON Btn_dummy;
        POSITION 50,100;
        LABEL "OFF";
        HIT ACTIONS;
            FETCH (Value) OFF;
            STORE Value;
        END ACTIONS;
    END BUTTON;

    BUTTON Btn_dummy;
        POSITION 150,100;
        LABEL "ON";
        HIT ACTIONS;
            FETCH (Value) ON;
            STORE Value;
        END ACTIONS;
    END BUTTON;
END PANEL;
```



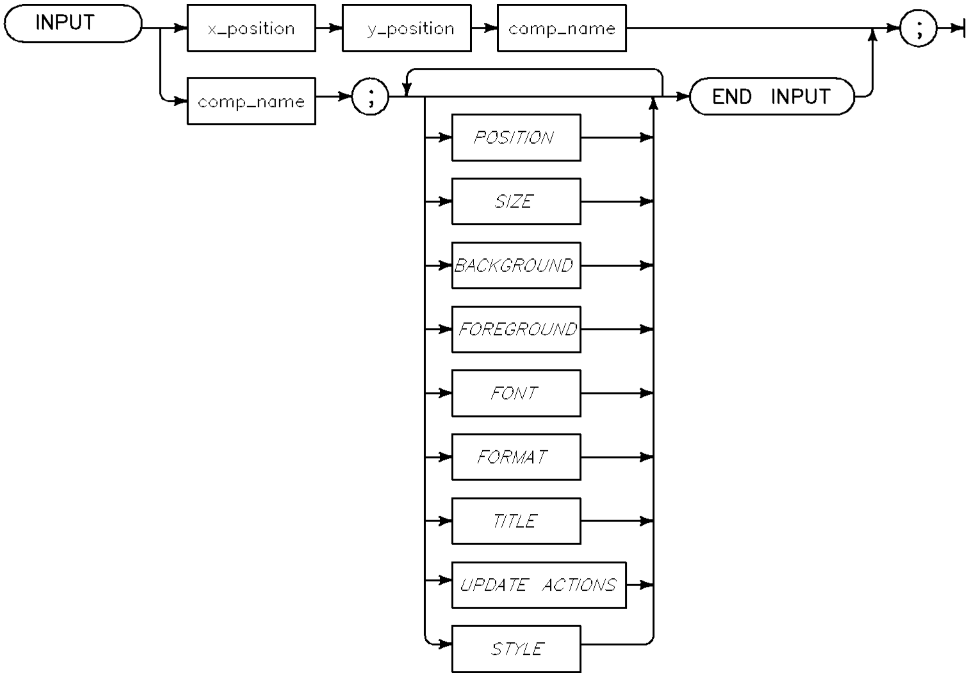
Cross Reference

- BUTTON
- DISPLAY
- XY

INPUT

The INPUT ... END INPUT compound statement creates a field on a soft panel in which the user can enter a value. The current value of an INPUT control is in this field. When the user clicks on the field, the development environment puts a cursor in the box, allowing the user to type in a new value. The user must press (Return) on the keyboard after the value is entered.

Syntax



Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the INPUT field.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the INPUT field.
comp_name	The name of an INTEGER, CONTINUOUS, or STRING component new value. The value entered is stored in this component.
POSITION	Consists of an x_position and a y_position that together specify the lower left corner of the INPUT field. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels of the characters associated with the panel element. These include the characters that constitute the current value and the TITLE. Default is 9 × 15 pixels.
FORMAT	Requires a string in which you insert either specifiers (see "FORMAT," table 9-4) or a number and the word DIGITS. The latter indicates that you want the current value rounded to the specified number of digits. Default is D.DDESZZ (for example, 4.33E+00).

(Continued)

INPUT

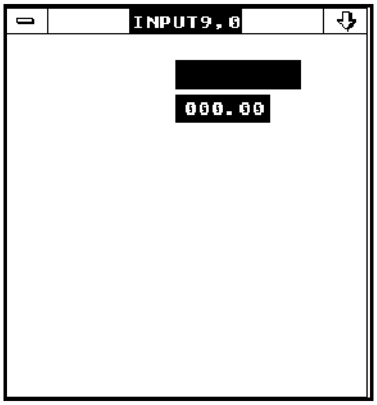
Item	Description

SIZE	Consists of a width and height that together specify the size in pixels of the input field. If not specified, the development environment computes a default size using the following formula:
	If the component is an INTEGER or a CONTINUOUS:
	width = (length of FORMAT × FONT width) + 4
	height = FONT height + 4
	If the component is a STRING;
	width = (length of FORMAT × FONT width) + 4
	height = FONT height + 4

BACKGROUND	Specifies the area fill color for the input field. Values must be provided for red, green, blue, and monochrome. Default color is French Gray (see "BACKGROUND," table 9-1) and black for monochrome displays.
FOREGROUND	Specifies the color of the text and border of the input field. Values must be provided for red, green, blue, and monochrome. Default color is White (see table 9-1) for color and monochrome displays.
TITLE	Specifies the text to be placed to the left of the input field. If you do not provide a TITLE, the development environment does not generate one. However, if you use the short form of the INPUT statement (that is, specify just the x_position, y_position, and comp_name), then the development environment generates a default TITLE, which is comp_name. The color of the characters is black.
UPDATE ACTIONS	An action list whose statements are executed whenever the view of the associated component is changed in the development environment.
STYLE	FILESELECT indicates that a filename will be required as input to the input control. The FILESELECT option is only valid with STRING components. FILESELECT uses a standard dialog box to let the user select filenames.

Example

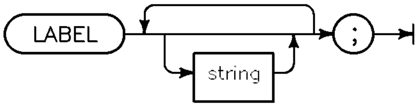
```
REVISION 2.0;

COMPONENT Astr;
    TYPE STRING 10;
END COMPONENT;

COMPONENT Anum;
    TYPE CONTINUOUS;
END COMPONENT;

PANEL Main;
    INPUT 100,180,Astr;

    INPUT Anum;
        POSITION 100,160;
        TITLE "Anum";
        FORMAT "ZZZ.DD";
    END INPUT;
END PANEL;
```



LABEL

LABEL is used to specify strings needed by DISCRETE, TOGGLE, and BUTTON controls.

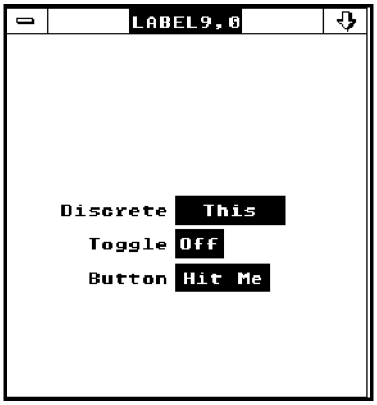
Syntax



Item	Description
string	For BUTTONs, **string** can contain: ■ A string that is the text to label this control on the panel. If not specified, the development environment uses the name of the associated component. For TOGGLEs, **string** can contain: ■ A string that is the text to label this control on the panel. If not specified, the development environment uses the information in the VALUES statement in the associated component. For DISCRETEs, **string** can contain: ■ Strings that are included in the list box from which the user selects. If not specified, the development environment uses the VALUES specified in the associated component.

Example

```
REVISION 2.0;

COMPONENT Btn;
   TYPE INTEGER;
END COMPONENT;

COMPONENT Sample;
    TYPE DISCRETE;
    VALUES A,B,C,D;
END COMPONENT;            ! Sample

PANEL Main;
    DISCRETE Sample;
        POSITION 100,100;
        LABEL "This","Is","An","Example";
        TITLE "Discrete";
    END DISCRETE;

    TOGGLE Sample;
        POSITION 100,80;
        LABEL "Off","On";
        TITLE "Toggle";
    END TOGGLE;

    BUTTON Btn;
        POSITION 100,60;
        LABEL "Hit Me";
        TITLE "Button";
    END BUTTON;
END PANEL;
```



Cross Reference

- BUTTON
- DISCRETE
- TOGGLE

MARKER

MARKER is used to specify a marker in an XY display. A marker can be positioned on a TRACE in the XY display, but it doesn't have to be. Markers that are not associated with a trace are called global markers.

There are three TYPEs of markers:

- A POINT (a diamond-shaped marker).
  - When used as a global marker, you must specify components that provide the x and y positions (see table 9-6).
  - When associated with a trace, you can specify that the marker's y position be derived from the data contained in the trace.
- A HORIZONTAL dotted line across the display.
  - The position of this marker is derived from the value of a component (comp_name).
- A VERTICAL dotted line from the top to the bottom of the display (comp_name).
  - The position of this marker is derived from the value of a component.

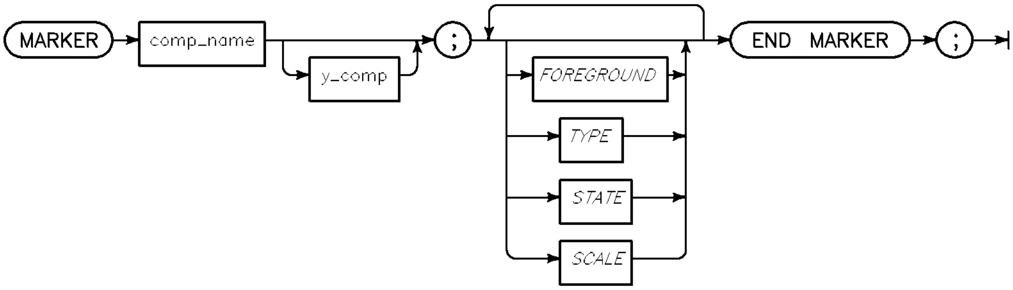
You can use the STATE statement to set up the MARKER so that it is displayed or not displayed depending on the value of a specified component.

You can use the SCALE statement to specify scaling for the MARKER that differs from the scaling for the TRACE or XY display.

Note

- If the positioning components are INVALID or DONTCARE, then the marker is not displayed.
- The development environment positions the MARKER in user units if SCALE is used (either in MARKER, TRACE, or XY), and pixels if SCALE is not in either the MARKER, TRACE, or XY display.

Syntax



Item	Description
comp_name	The name of a CONTINUOUS or INTEGER component whose value HP ITG uses to determine the position of the marker, depending on the TYPE of marker specified (see table 9-6).
y_comp	The name of a CONTINUOUS or INTEGER component whose value the development environment uses to determine the y position of a TYPE POINT marker; y_comp is ignored for TYPE VERTICAL and HORIZONTAL.
FOREGROUND	Specifies the color of the marker. The default color is White for both color and monochrome displays.
TYPE	Specifies the type of marker (that is, POINT, HORIZONTAL, or VERTICAL). Default is POINT.
STATE	Specifies whether the marker is displayed. The default is to display the marker.
SCALE	Specifies the user units the development environment uses when positioning the MARKER, if they differ from the TRACE or XY SCALE.

Table 9-7. Relationship of MARKER TYPE with comp_name

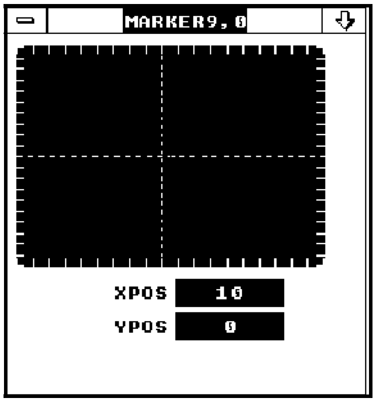
MARKER TYPE	y_comp specified?	Is MARKER in TRACE?	How comp_name is used
POINT	Yes	Don't care	The x-position of the marker, in user units (see "SCALE" in this chapter).
POINT	No	Yes	The offset into the array containing the trace data *and* the x-position in user units.
POINT	No	No	This is an error.
HORIZONTAL	Don't care	Don't care	The y-position of the marker in user units (see "SCALE" in this chapter).
VERTICAL	Don't care	Don't care	The x-position of the marker in user units (see "SCALE" in this chapter).

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
        VALIDATE XYDATA;
    END ACTIONS;
END COMPONENT;                    ! Reset

COMPONENT Xydata;
    TYPE RARRAY 2,20;
END COMPONENT;                    ! Xydata

COMPONENT Xpos;
    TYPE CONTINUOUS;
    VALUES RANGE 1,20,1;
    INITIAL 10;
END COMPONENT;                    ! Xpos

COMPONENT Ypos;
    TYPE CONTINUOUS;
    VALUES RANGE -10,10,1;
    INITIAL 0;
END COMPONENT;                    !Ypos

PANEL Main;
    XY;
        SCALE 1,20,-10,10;
        GRATICULE FRAME;
        MARKER Xpos;
            TYPE VERTICAL;
        END MARKER;
```

(Continued)

```
           MARKER Ypos;
               TYPE HORIZONTAL;
           END MARKER;
           MARKER Xpos,Ypos;
               TYPE POINT;
               FOREGROUND (230,30,30),100;
           END MARKER;
           TRACE Xydata;
               MARKER Xpos;
                   FOREGROUND (120,184,210),100;
               END MARKER;
           END TRACE;
       END XY;

       CONTINUOUS 100,50,Xpos;
       CONTINUOUS 100,30,Ypos;
   END PANEL;                      ! Main
```



MARKER9,0

XPOS 10

YPOS 0

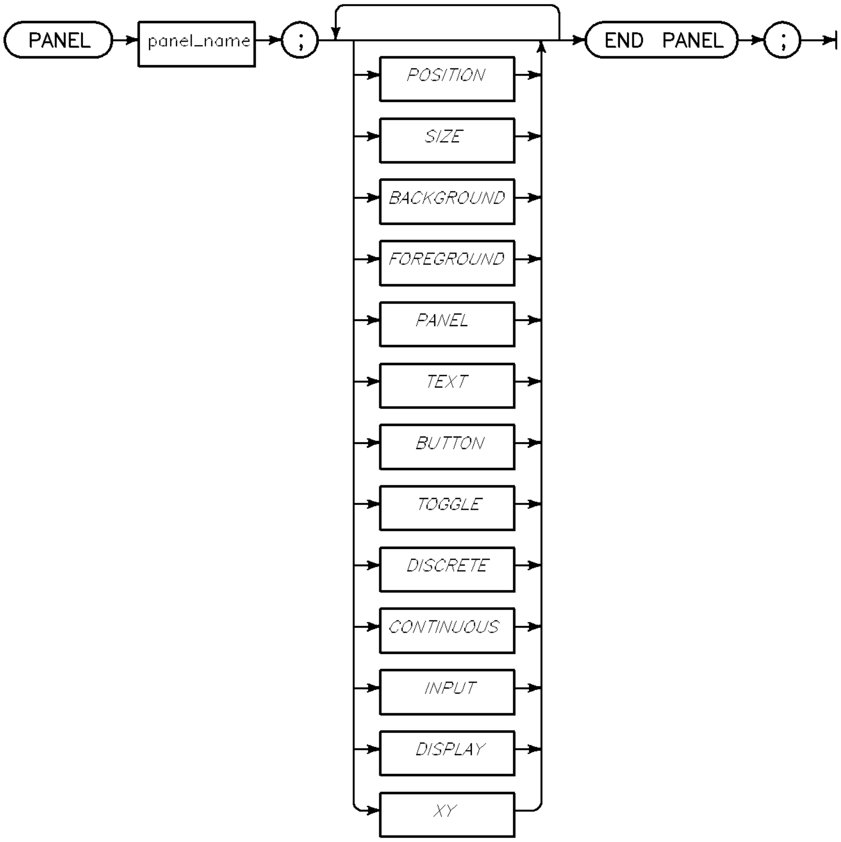
Cross Reference

- SCALE
- STYLE
- TRACE
- TYPE
- XY

PANEL

The PANEL ... END PANEL compound statement is used to specify the layout of an entire soft panel as well as any subpanels.

Syntax



PANEL

Item	Description
panel_name	This name is used to reference the panel in SHOW and HIDE action statements (see “SHOW” and “HIDE” in chapter 8, “Component/Action Syntax”). Each panel must have a unique name.
POSITION	Specifies the lower left corner of a subpanel relative to the main panel. The development environment determines the position of an HP ID’s main panel, and so ignores any POSITION specified. Default is 6,6 for subpanels, meaning 6 pixels over and 6 pixels up from the bottom left corner of the main panel.
SIZE	Specifies the size of the panel in pixels. Default is 214 × 213 pixels for the main panel, and 202 × 176 pixels for subpanels.
BACKGROUND	Specifies the area fill color of the panel. If not specified for a main panel, default is Parchment White (see “BACKGROUND,” table 9-1), black for monochrome displays. If not specified for a subpanel, default is the color of the parent panel.
FOREGROUND	Specifies the color of the edge of the panel. If not specified for a main panel, default is White (see table 9-1) for color and monochrome displays. If not specified for a subpanel, the subpanel does not have an edge drawn.
PANEL	Allows you to create subpanels by nesting PANEL statements.
TEXT	Used to create text on the panel. Most often used to label controls and displays.
BUTTON	Used to create a pushbutton control.
TOGGLE	Used to create a button that toggles between two values.
DISCRETE	Used to create a control in which the user selects a value from a list of allowable values.

(Continued)

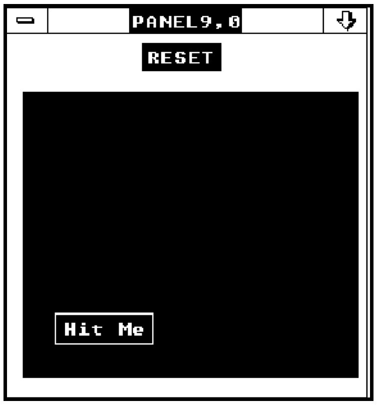
Item	Description
CONTINUOUS	Used to create a control in which the user can enter a numeric value.
INPUT	Used to create a field in which the user can type in a value.
DISPLAY	Used to create a field in which the measurement or reading can be displayed.
XY	Used to create an XY display.

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;
COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        SHOW Second;
    END ACTIONS;
END COMPONENT;            ! Reset

COMPONENT Dmybtn;
    TYPE INTEGER;
END COMPONENT;            ! Dmybtn

PANEL Main;
    BUTTON 80,190,Reset;
    PANEL Second;
        POSITION 10,10;
        SIZE 200,170;
        BACKGROUND (120,184,210),100;
```

(Continued)

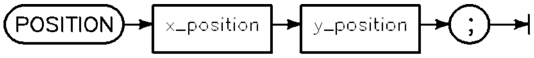
```
            BUTTON Dmybtn;
                POSITION 20,20;
                LABEL "Hit Me";
                HIT ACTIONS;
                    HIDE Second;
                END ACTIONS;
            END BUTTON;
         END PANEL;
    END PANEL;
```



POSITION

Specifies the lower left corner of a panel element (that is, text, display, or control) relative to the lower left corner of the panel containing the element. The development environment determines the POSITION for the main panel, and so ignores any POSITION specified. Refer to each panel element for default information.

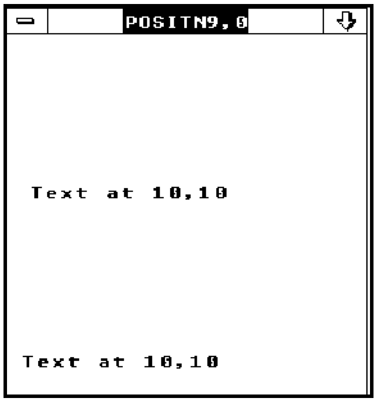
Syntax



Item	Description
x_position	An integer specifying the number of pixels to the right of the lower left corner of the panel.
y_position	An integer specifying the number of pixels above the lower left corner of the panel.

Example

```
REVISION 2.0;

INITIALIZE Startup;

COMPONENT Startup;
    TYPE INTEGER;
    SET ACTIONS;
        SHOW Sub;
    END ACTIONS;
END COMPONENT;

PANEL Main;

    TEXT 10,10,"Text at 10,10";

    PANEL Sub;
        POSITION 5,100;
        SIZE 200,100;
        FOREGROUND (230,30,30),100;

        TEXT 10,10,"Text at 10,10";
    END PANEL;
END PANEL;
```



Text at 10,10

Text at 10,10

Cross Reference

- BUTTON
- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- PANEL
- TEXT
- TOGGLE
- XY

SCALE

SCALE is used to define the unit of measure for the different elements of an XY display. The parameters of SCALE specify the values at the boundaries of the XY display.

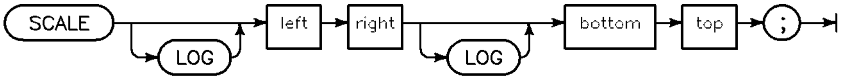
SCALE can be specified for XY displays, TRACEs, and MARKERs.

The SCALE specified in the XY display applies to the GRATICULE and any TRACEs or global MARKERS in the XY display that do not have their own SCALE. If SCALE is not specified in an XY display, then pixels are used as the unit of measure with 0,0 being the lower left corner.

A SCALE specified within a TRACE applies to the trace and any MARKERS defined *following* the SCALE statement within the TRACE. If no SCALE is specified for a TRACE, then the SCALE for the XY display is used.

A SCALE specified for a MARKER applies only to that MARKER. If no SCALE is specified for the MARKER, then the SCALE for the TRACE of the XY display in which it is declared will be used.

Syntax



A LOG (base 10) on the scale does two things. First, the scale value pairs, either left/right or bottom/top, are passed to the trace drawing routines as the log of the values. Second, if LOG is specified in front of the left/right pair, then all the X values are converted to the log of the values. If LOG is specified in front of the bottom/top pair, then all the Y values of the graph are converted to the log of the Y values.

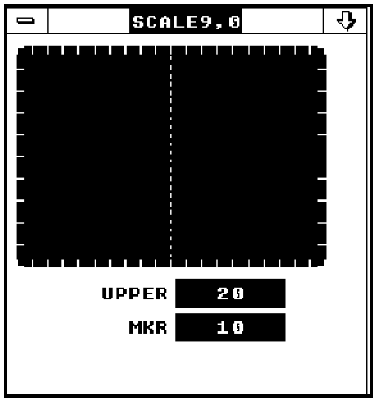
Item	Description
left	A real number or name of an INTEGER or CONTINUOUS component whose value specifies the minimum data value for the X axis.
right	A real number or name of an INTEGER or CONTINUOUS component whose value specifies the maximum data value for the X axis.
bottom	A real number or name of an INTEGER or CONTINUOUS component whose value specifies the minimum data value for the Y axis.
top	A real number or name of an INTEGER or CONTINUOUS component whose value specifies the maximum data value for the Y axis.
LOG	Specifies that the graph will be a logarithmic axis on one or both axes. LOG is taken as the base 10 logarithm.

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
        VALIDATE Xydata;
    END ACTIONS;
END COMPONENT;              ! Reset
```

(Continued)

```
COMPONENT Xydata;
    TYPE RARRAY 2,20;
END COMPONENT;               ! Xydata

COMPONENT Upper;
    TYPE CONTINUOUS;
    VALUES RANGE 1,100,1;
    INITIAL 20;
END COMPONENT;               ! Upper

COMPONENT Mkr;
    TYPE CONTINUOUS;
    VALUES RANGE 1,100,1;
    INITIAL 10;
END COMPONENT;               ! Mkr;

PANEL Main;
    XY;
        SCALE 0,Upper,-5,5;
        GRATICULE FRAME;
        MARKER Mkr;
            TYPE VERTICAL;
        END MARKER;
        TRACE Xydata;
 END TRACE;
    END XY;

    CONTINUOUS 100,50,Upper;
    CONTINUOUS 100,30,Mkr;
END PANEL;
```



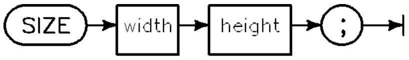
Cross Reference

- GRATICULE
- MARKER
- TRACE
- XY

SIZE

SIZE specifies the dimensions of a panel element (text, display, or control) in pixels.

Syntax



Item	Description
width	An integer specifying the width in pixels of a panel element.
height	An integer specifying the height in pixels of a panel element.

The development environment determines the default size of a panel element using the following formulas:

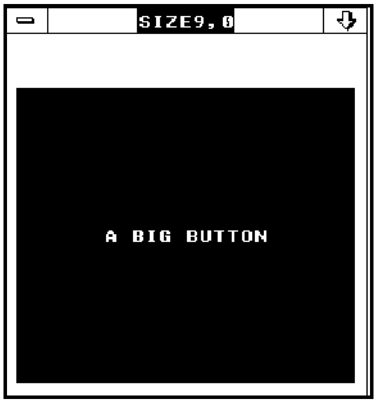
$$\text{width} = ((\text{LEN(LABEL)} \times \text{font\_width}) + 4$$
$$\text{height} = \text{font\_height} + 4$$

Example

```
REVISION 2.0;

COMPONENT Btn;
    TYPE INTEGER;
    SET ACTIONS;
        NOTIFY "The button was pressed.";
    END ACTIONS;
END COMPONENT;
```

(Continued)

```
PANEL Main;
    BUTTON Btn;
        POSITION 6,6;
        SIZE 202,176;
        LABEL "A BIG BUTTON";
    END BUTTON;
END PANEL;
```



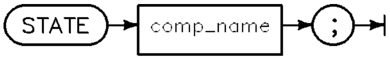
Cross Reference

- BUTTON
- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- TEXT
- TOGGLE

STATE

STATE specifies whether a MARKER or TRACE is displayed.

Syntax



Item	Description
comp_name	The name of an INTEGER, CONTINUOUS, or DISCRETE component whose value determines whether a marker or trace is displayed. For INTEGER or CONTINUOUS components: ■ If the value is 0, the marker or trace is not displayed. Otherwise, the marker or trace is displayed. For DISCRETE components: ■ If the current value is the first value in the list, then the marker or trace is not displayed. Otherwise, the marker or trace is displayed.

Example

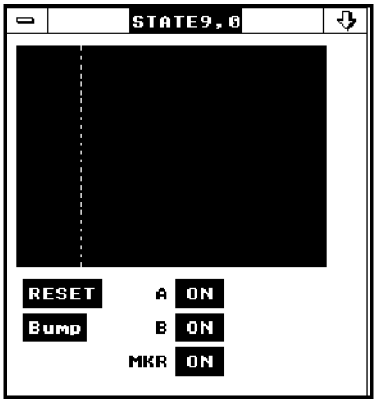
```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
        VALIDATE Xy_a;
        VALIDATE Xy_b;
        MATSCALE 0,1,Xy_a;
    END ACTIONS;
END COMPONENT;                    ! Reset

COMPONENT Xy_a;
    TYPE RARRAY 1,20;
END COMPONENT;                    ! Xy_a

COMPONENT Xy_b;
    TYPE RARRAY 1,20;
END COMPONENT;                    ! Xy_b

COMPONENT Mrk_val;
    TYPE CONTINUOUS;
    VALUES RANGE 1,20;
    INITIAL 5;
END COMPONENT;                    ! Mkr_val

COMPONENT A;
    TYPE DISCRETE;
    VALUES OFF,ON;
    INITIAL ON;
END COMPONENT;                    ! A
```

(Continued)

```
COMPONENT B;
    TYPE DISCRETE;
    VALUES OFF,ON;
    INITIAL ON;
END COMPONENT;                   ! B

COMPONENT Mkr;
    TYPE DISCRETE;
    VALUES OFF,ON;
    INITIAL ON;
END COMPONENT;                   ! Mkr

COMPONENT Btn;
    TYPE INTEGER;
END COMPONENT;                   ! Btn

PANEL Main;
    XY;
        SCALE 1,20,-5,5;
        MARKER Mrk_val;
            TYPE VERTICAL;
            STATE Mrk;
        END MARKER;
        TRACE Xy_a;
            STATE A;
        END TRACE;
        TRACE Xy_b;
            STATE B;
        END TRACE;
    END XY;

    TOGGLE 100,50,A;
    TOGGLE 100,30,B;
    TOGGLE 100,10,Mrk;
```

(Continued)

```
        BUTTON 10,50,Reset;

        BUTTON Btn;
            POSITION 10,30;
            LABEL "Bump";
            HIT ACTIONS;
                MATSCALE 1,1,Xy_a;
            END ACTIONS;
        END BUTTON;
    END PANEL;
```



Cross Reference

- MARKER
- TRACE

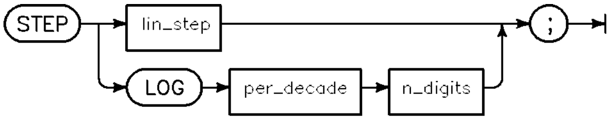
STEP

STEP specifies the amount by which the up/down arrows on the ends of the scroll bar in a CONTINUOUS dialog box change the displayed value. STEP can be specified to increase linearly or logarithmically.

If STEP is specified, the resolution is continuous. If STEP is not specified, the resolution from the associated component is used to determine the step size (see "VALUES" in chapter 8, "Component/Action Syntax").

Note Do not use STEP LOG if the upper or lower limit of the VALUES RANGE of the component is less than or equal to 0.

Syntax



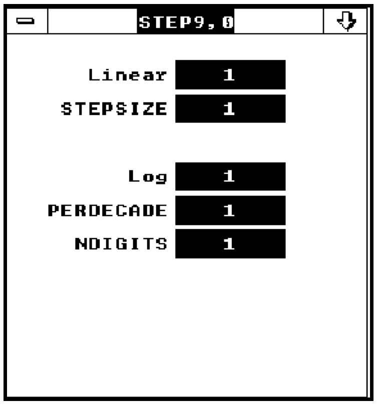
Item	Description
lin_step	A number or name of an INTEGER or a CONTINUOUS component whose value is used as the step size.
per_decade	An integer or name of a INTEGER component whose value specifies the number of steps per decade that the development environment should increment the current value.
n_digits	An integer or name of a INTEGER component whose value specifies the number of digits of resolution. To increase by decade (for example, 3, 30, 300): set **per_decade** = 1 and **n_digits** = 1 To increase in a 1-3 sequence: set **per_decade** = 2 and **n_digits** = 1 To increase in a 1-2-5 sequence: set **per_decade** = 3 and **n_digits** = 1

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
    END ACTIONS;
END COMPONENT;

COMPONENT Linstep;
    TYPE CONTINUOUS;
    VALUES RANGE -10,10;
    INITIAL 0;
END COMPONENT;
```

(Continued)

```
COMPONENT Logstep;
    TYPE CONTINUOUS;
    VALUES RANGE 1,1E12;
    INITIAL 0;
END COMPONENT;

COMPONENT Stepsize;
    TYPE CONTINUOUS;
    VALUES RANGE 0,10;
    INITIAL 1;
END COMPONENT;

COMPONENT Perdecade;
    TYPE INTEGER;
    VALUES RANGE 1,20;
    INITIAL 1;
END COMPONENT;

COMPONENT Ndigits;
    TYPE INTEGER;
    VALUES RANGE 1,16;
    INITIAL 1;
END COMPONENT;

PANEL Main;
    CONTINUOUS Linstep;
        POSITION 100,180;
        STEP Stepsize;
        TITLE "Linear";
    END CONTINUOUS;

    CONTINUOUS 100,160,Stepsize;

    CONTINUOUS Logstep;
        POSITION 100,120;
        STEP LOG Perdecade,Ndigits;
        TITLE "Log";
    END CONTINUOUS;

    CONTINUOUS 100,100,Perdecade;
    CONTINUOUS 100,80,Ndigits;
END PANEL;
```



STEP9,0

Linear	1
STEPSIZE	1
Log	1
PERDECADE	1
NDIGITS	1

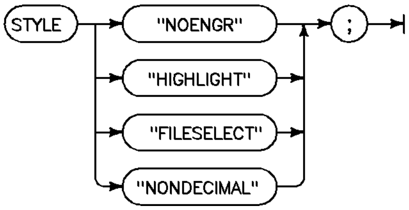
Cross Reference

- CONTINUOUS

STYLE

STYLE specifies extra parameters on CONTINUOUS, DISPLAY, and TOGGLE. Please refer to these elements for a complete description of how STYLE can be used.

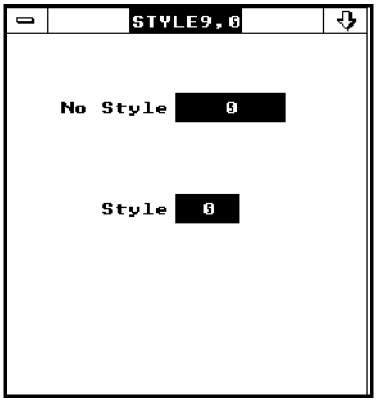
Syntax



Item	Description
NOENGR	For CONTINUOUS controls and DISPLAYs, values are displayed without engineering suffixes.
HIGHLIGHT	For TOGGLE controls, the toggle button's color changes as follows: ■ First toggle value in list: □ Text Color is set by FOREGROUND. □ Area Fill color is set by BACKGROUND. ■ Any other toggle value: □ Text Color is set by BACKGROUND. □ Area Fill color is set by FOREGROUND.
NONDECIMAL	No suffixes are displayed. Instead, the entry box contains additional field for binary, octal, decimal, and hexadecimal entry. The display format is controlled by the FORMAT specifier.

Example

```
REVISION 2.0;

COMPONENT Value;
    TYPE CONTINUOUS;
END COMPONENT;

PANEL Main;
    CONTINUOUS Value;
        POSITION 100,160;
        TITLE "No Style";
    END CONTINUOUS;

    CONTINUOUS Value;
        POSITION 100,100;
        TITLE "Style";
        STYLE "NOENGR";
    END CONTINUOUS;
END PANEL;
```



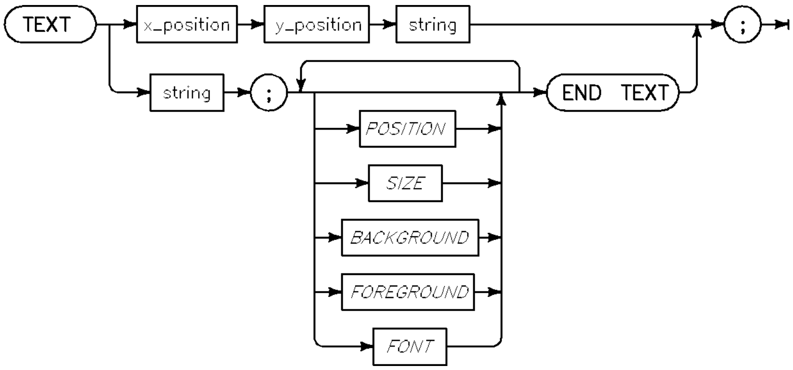
Cross Reference

- CONTINUOUS
- DISPLAY
- INPUT
- TRACE
- TOGGLE

TEXT

The TEXT ... END TEXT compound statement is used to create text on the panel. The text is most often used to label controls and displays.

Syntax



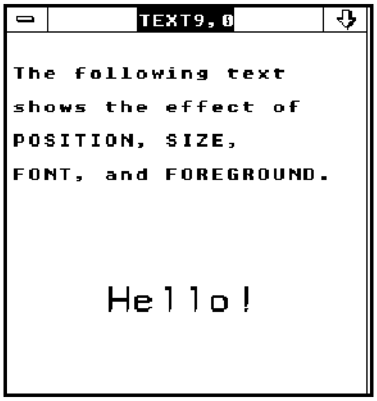
Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the TEXT.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the TEXT.
string	This is the actual text that is displayed on the panel as specified by the other parameters.
POSITION	Consists of an x-position and a y_position that together specify the lower left corner of the TEXT. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels of the characters in TEXT. Default is 9 × 15 pixels.

SIZE	Consists of a width and height that together specify the size in pixels of the TEXT. If not specified, the development environment computes a default size using the following formula:
	width = length of string × FONT width
	height = FONT height

BACKGROUND	The BACKGROUND statement is present for compatibility but has no affect on TEXT background color. TEXT uses the panel background color.
FOREGROUND	Specifies the color for the TEXT. Values must be provided for red, green, blue, and monochrome. Default color is black for color displays and white for monochrome displays.

Example

```
REVISION 2.0;

PANEL Main;

    TEXT 5,180,"The following text";
    TEXT 5,160,"shows the effect of";
    TEXT 5,140,"POSITION, SIZE";
    TEXT 5,120,"FONT, and FOREGROUND.";

    TEXT "Hello!";
        POSITION 5,5;
        SIZE 200,100;
        FONT 15,25;
        FOREGROUND (0,0,0),0;
    END TEXT;
END PANEL;
```



```
The following text
shows the effect of
POSITION, SIZE,
FONT, and FOREGROUND.

Hello!
```

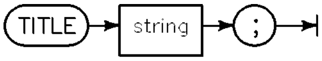
Cross Reference

- TITLE

TITLE

TITLE specifies text that is positioned to the left of some displays and controls. This is an easy way to label controls instead of using TEXT. The text in the TITLE is displayed in black on color displays and white on monochrome displays.

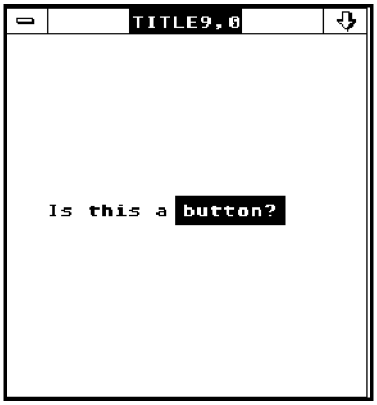
Syntax



Item	Description
string	Text that is displayed 3 pixels to the left of the control or display.

Example

```
REVISION 2.0;

COMPONENT Btn;
    TYPE INTEGER;
    SET ACTIONS;
        NOTIFY "Yup!";
    END ACTIONS;
END COMPONENT;

PANEL Main;
    BUTTON Btn;
        POSITION 100,100;
        TITLE "Is this a";
        LABEL "button?";
    END BUTTON;
END PANEL;
```



Cross Reference

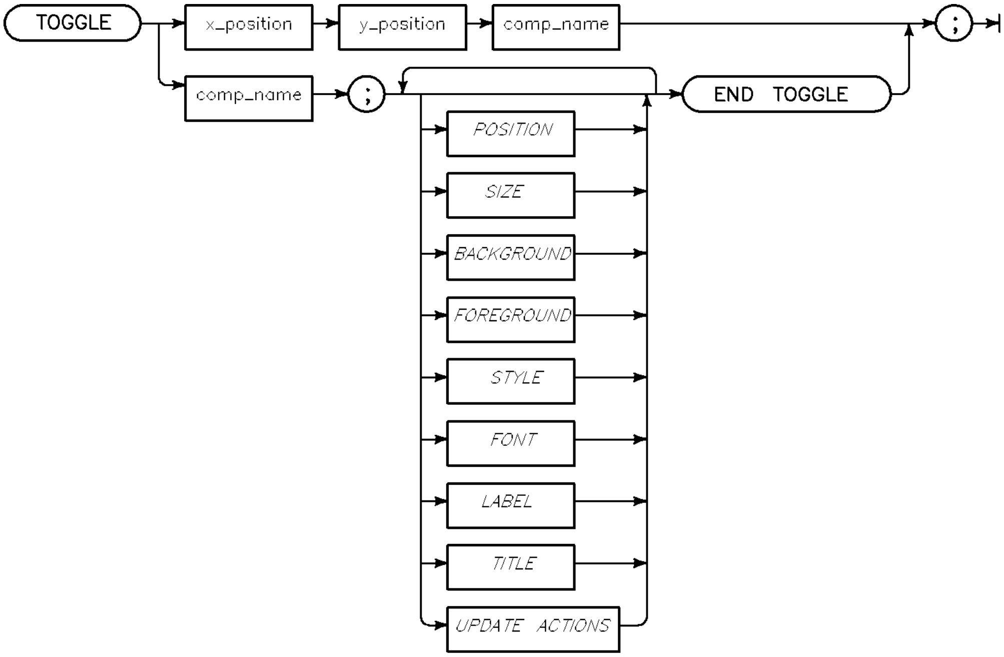
- BUTTON
- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- TOGGLE

TOGGLE

The TOGGLE ... END TOGGLE compound statement is used to create a button that toggles between two values.

When the user clicks on a toggle button in the development environment, it executes the SET ACTIONS in the DISCRETE component with which the toggle is associated.

Syntax



TOGGLE

Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the toggle.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the toggle.
comp_name	The name of a DISCRETE component whose SET ACTIONS are executed after the user has toggled to a new value. The component should have a list of two values (see "VALUES" in chapter 8, "Component/Action Syntax"). The first value is the initial value, unless an INITIAL statement is used (see "INITIAL" in chapter 8, "Component/Action Syntax").
POSITION	Consists of an x-position and a y-position that together specify the lower left corner of the toggle. The default is 1,1, meaning 1 pixel over and 1 pixel up from the bottom left corner of the panel.
FONT	Consists of a width and height that together specify the size in pixels of the characters in toggle. Default is 9 × 15 pixels.

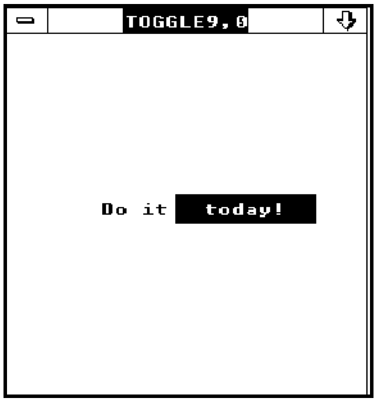
SIZE	Consists of a width and height that together specify the size in pixels of the toggle. If not specified, the development environment computes a default size using the following formula:
	width = (length of longest LABEL × FONT width) + 4
	height = FONT height + 4

(Continued)

Item	Description
STYLE	If string = "HIGHLIGHT", then the color of the toggle button changes as follows: ■ First toggle value in list: □ Text Color is set by FOREGROUND. □ Area Fill color is set by BACKGROUND. ■ Any other toggle value: □ Text Color is set by BACKGROUND. □ Area Fill color is set by FOREGROUND.
BACKGROUND	Specifies the area fill color for the toggle. Values must be provided for red, green, blue, and monochrome. Default color is Evening Gold (see "BACKGROUND," table 9-1), and black for monochrome displays.
FOREGROUND	Specifies the color of the text and border of the toggle. Values must be provided for red, green, blue, and monochrome. Default color is White (see table 9-1) for color and monochrome displays.
LABEL	Specifies the text that appears in the toggle. The first string is the text when the toggle is off (that is, the current value is the first value in the DISCRETE component VALUES list), and the second string is the text when the toggle is on. If not specified, the development environment uses the VALUES from the component (see "VALUES" in chapter 8, "Component/Action Syntax").
TITLE	Specifies the text to be placed to the left of the toggle. If not specified, the development environment does not generate one. However, if you use the short form of the TOGGLE statement (that is, specify just the x_position, y_position, and comp_name), then the development environment generates a default TITLE, which is comp_name.
UPDATE ACTIONS	An action list that the development environment executes whenever the value of the component is changed in the development environment.

Example

```
REVISION 2.0;

COMPONENT Tgl;
    TYPE DISCRETE;
    VALUES NO,YES;
END COMPONENT;

PANEL Main;
    TOGGLE Tgl;
        POSITION 100,100;
        LABEL "today!","tomorrow!";
        TITLE "Do it";
        STYLE "HIGHLIGHT";
    END TOGGLE;
END PANEL;
```



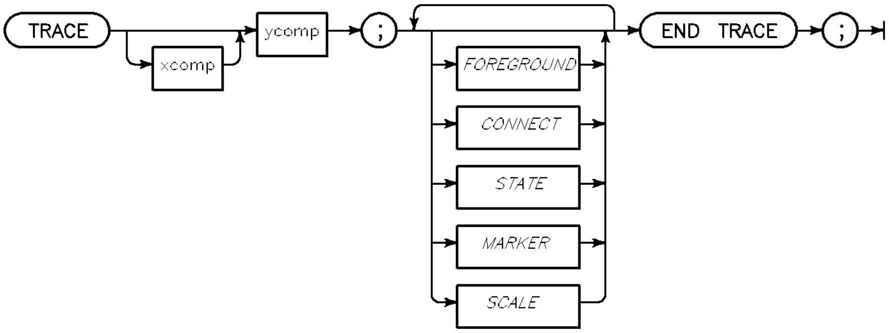
TRACE

TRACE is used to specify details about a trace on an XY display. You need to use a TRACE statement for every trace you want displayed. The number of traces allowed is limited only by your system's memory.

The development environment provides for having an optional X component on the TRACE statement. This allows you to create a trace with the X-axis data specified in one component and the Y-axis data in the same or different component (see the following syntax diagram).

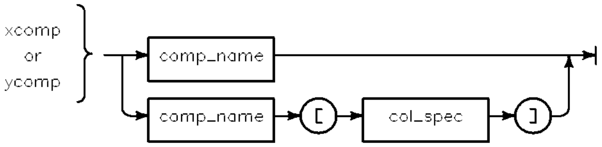
In addition, the X and Y components can optionally specify which row/column of data to be used in the plot. The row/column information is defined differently depending if the array is a 1-dimension or 2-dimension array.

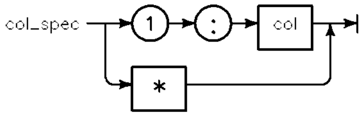
Syntax



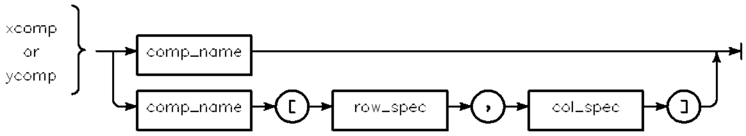
If the Xcomp is given, it must resolve down to a vector or unpredictable results may occur. Only one row of data can be allowed for the X axis. Multiple Y rows yield multiple plots of the same color.

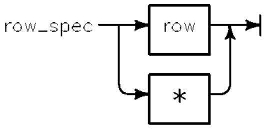
1-Dimension Array Component

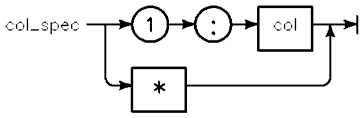




2-Dimension Array Component







Item	Description
xcomp	See drawing for 1- or 2-dimension array component.
ycomp	See drawing for 1- or 2-dimension array component.
FOREGROUND	Specifies the color of the TRACE. Default colors are listed in table 9-7, following page. Default for monochrome is a white trace.
CONNECT	Specifies whether the trace data points should be connected. Default is connected.
STATE	Specifies whether the TRACE is displayed. The default is to display the TRACE.
MARKER	Specifies a marker that is associated with the TRACE. Any TYPE of MARKER can be defined in a TRACE, but POINT markers are the only TYPE that use trace data. Default is no markers.
SCALE	Specifies the user unit scaling of the TRACE. SCALE requires the parameters left, right, top, and bottom for the minimum and maximum data values for the x and y axes respectively. Default is 1 unit per pixel, with left and bottom being zero.
comp_name	The name of an IARRAY or RARRAY component that contains data to be plotted on the XY display.
col_spec	See drawing for col_spec.
row_spec	See drawing for row_spec.
col	A real number or name of an INTEGER or CONTINUOUS component whose value specifies the column to be used.
row	A real number or name of an INTEGER or CONTINUOUS component whose value specifies the row to be used.
*	Use the character, *, which means to use the whole row or column of data.

Example

Here are some examples of array row/columns specifications and example TRACE statements. These will be inside a TRACE statement as either the X or Y component:

1-dimension:

`data1[1:Numpts]`	Plot the first <Numpts> pts. Numpts is numeric component.
`data1[1:100]`	Plot the first 100 pts.
`data1[*]`	Plot the whole vector.
`data1`	Plot the whole vector.

```
TRACE data1[1:Numpts] data2[2,1:Numpts];
END TRACE;
```

2-dimension:

`data2`	Multi plot all rows as same color.
`data2[*,*]`	ERROR - invalid ROW_SPEC
`data2[1]`	ERROR - must have row/col specifier.
`data2[1,*]`	Single plot of first row of data.
`data2[3,1:50]`	Plot first 50 points from row 3.

```
TRACE data2;
END TRACE;
```

Table 9-8. Default Trace Colors

Trace	Color	Red	Green	Blue	Mono
First	Yellow	255	255	0	100
Second	Cyan	0	255	255	100
Third	Magenta	255	0	255	100
Fourth	Rich Green	0	200	50	100
Other	White	255	255	255	100

Cross Reference

- MARKER
- XY

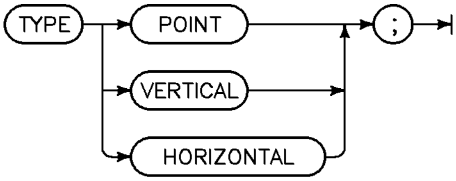
TYPE

TYPE specifies the type of MARKER to be displayed. There are three types of markers:

- A POINT (a diamond-shaped marker).
- A HORIZONTAL dotted line across the display.
- A VERTICAL dotted line from the top to the bottom of the display

If TYPE is not specified, POINT is the default.

Syntax



Item	Description
POINT	When used as a global marker, you must specify components that provide the x and y positions (see table 9-8). When associated with a trace, you can specify that the marker's y position be derived from the data contained in the trace.
VERTICAL	The position of this marker is derived from the value of a component (comp_name).
HORIZONTAL	The position of this marker is derived from the value of a component (comp_name).

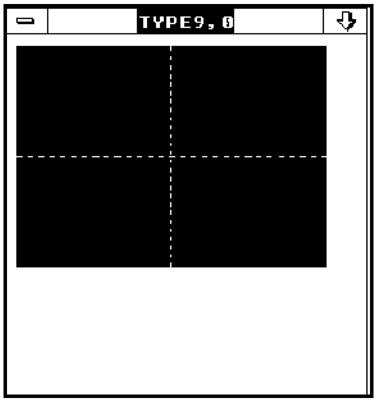
Table 9-9. Relationship of MARKER TYPE with comp_name

MARKER Type	y_comp Specified?	Is MARKER in TRACE?	How comp_name is used
POINT	Yes	Don't care	The x-position of the marker, in user units (see "SCALE" in this chapter).
POINT	No	Yes	The offset into the array containing the trace data and the x-position in user units.
POINT	No	No	This is an error.
HORIZONTAL	Don't care	Don't care	The y-position of the marker in user units (see "SCALE" in this chapter).
VERTICAL	Don't care	Don't care	The x-position of the marker in user units (see "SCALE" in this chapter).

Example

```
REVISION 2.0;

COMPONENT X;
    TYPE CONTINUOUS;
END COMPONENT;
COMPONENT Y;
    TYPE CONTINUOUS;
END COMPONENT;
PANEL Main;
    XY;
        SCALE -10,10,-10,10;
        MARKER X;
            TYPE VERTICAL;
        END MARKER;
        MARKER Y;
            TYPE HORIZONTAL;
        END MARKER;
    END XY;
END PANEL;
```

TYPE



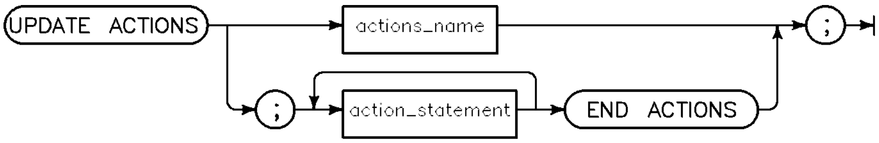
Cross Reference

- GRATICULE
- MARKER
- TRACE
- XY

UPDATE ACTIONS

UPDATE ACTIONS are a set of action statements that are executed whenever the value of the associated component changes.

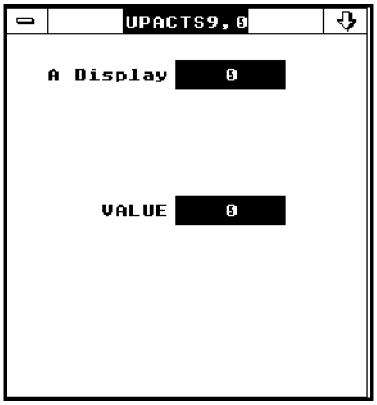
Syntax



Item	Description
action_statement	See tables 8-1 through 8-8 under "ACTIONS" in chapter 8, "Component/Action Syntax" for valid action statements.
actions_names	The name of the action list.

Example

```
REVISION 2.0;

COMPONENT Value;
    TYPE CONTINUOUS;
    SET ACTIONS;
        NOTIFY "Set Actions.";
    END ACTIONS;
    GET ACTIONS;
        NOTIFY "Get Actions.";
    END ACTIONS;
    PANEL SET ACTIONS;
        NOTIFY "Panel Set Actions.";
    END ACTIONS;
END COMPONENT;

PANEL Main;
    DISPLAY Value;
        POSITION 100,180;
        TITLE "A Display";
        UPDATE ACTIONS;
            NOTIFY "Update Actions.";
        END ACTIONS;
    END DISPLAY;

    CONTINUOUS 100,100,Value;
END PANEL;
```



UPACTS9,0

A Display 0

VALUE 0

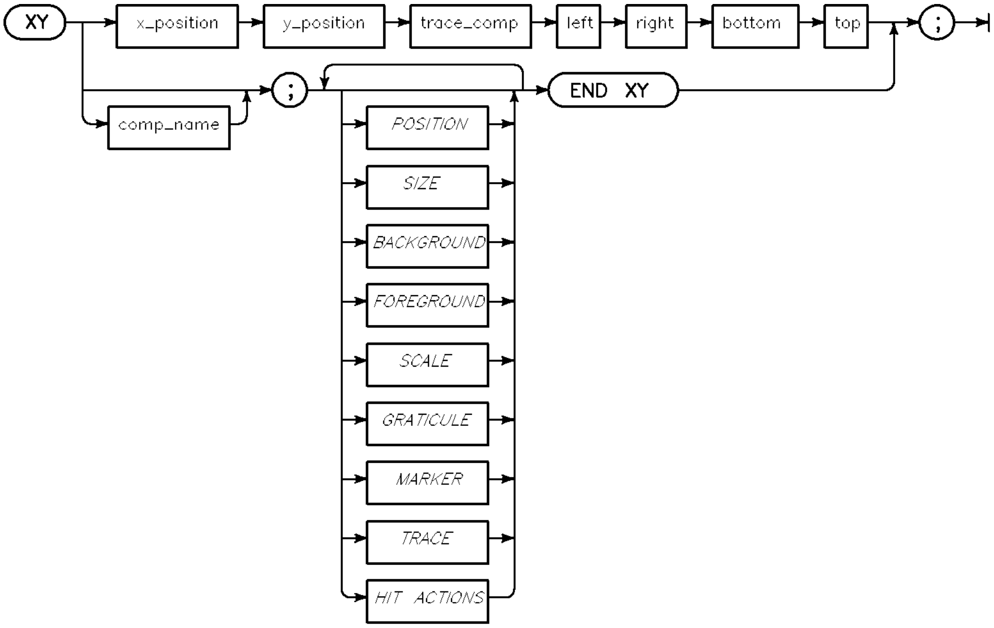
Cross Reference

- CONTINUOUS
- DISCRETE
- DISPLAY
- INPUT
- TOGGLE

XY

The XY ... END XY compound statement is used to create an XY display on a panel. An XY display can plot data from a real or integer array (that is, from a RARRAY or IARRAY component).

Syntax



Item	Description
x_position	An integer specifying the number of pixels to the right of the bottom left corner of the panel. Used with y_position to specify the bottom left corner of the XY display.
y_position	An integer specifying the number of pixels above the bottom left corner of the panel. Used with x_position to specify the bottom left corner of the XY display.
trace_comp	The name of an IARRAY or RARRAY component that contains the data to be plotted on the XY display.
left	A number or name of a CONTINUOUS or INTEGER component whose value specifies the left data point of the XY display (see "SCALE" in this chapter).
right	A number or name of a CONTINUOUS or INTEGER component whose value specifies the right data point of the XY display (see "SCALE" in this chapter).
bottom	A number or name of a CONTINUOUS or INTEGER component whose value specifies the bottom data point of the XY display (see "SCALE" in this chapter).
top	A number or name of a CONTINUOUS or INTEGER component whose value specifies the top data point of the XY display (see "SCALE" in this chapter).
comp_name	The name of an INTEGER component whose GET ACTIONS are executed when the user clicks on the XY display (see "HIT ACTIONS" in this chapter). If comp_name is not specified and there are no HIT ACTIONS, then GET ACTIONS are executed on each TRACE and MARKER in the XY display.

(Continued)

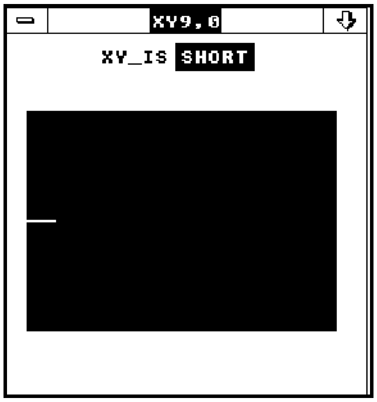
Item	Description
POSITION	Consists of an x-position and a y-position that together specify the lower left corner of the XY display. The default is 6,74, meaning 6 pixels over and 74 pixels up from the bottom left corner of the panel.
SIZE	Consists of a width and height that together specify the size in pixels of the XY display. Default is 185 x133.
BACKGROUND	Specifies the area fill color for the XY display. Values must be provided for red, green, blue, and monochrome. Default color is French Gray (see “BACKGROUND,” table 9-1), and black for monochrome displays.
FOREGROUND	Specifies the color of the border of the XY display. Values must be provided for red, green, blue, and monochrome. Default color is White (see table 9-1) for color and monochrome displays.
SCALE	Specifies the user unit scaling of the XY display. SCALE requires the parameters left, right, top, and bottom for the minimum and maximum data values for the x and y axes respectively. Default is 1 unit per pixel, with left and bottom being zero.
GRATICULE	Specifies the graticule that the development environment plots on the XY display. Default is no graticule. Allowable values are AXES, GRID, SMITH, INVSMITH, POLAR, FRAME, and comp_name.
MARKER	Specifies a global marker (that is, not associated with a TRACE). You must specify a y-component for a POINT global MARKER.
TRACE	Specifies the traces to be plotted on the XY display.
HIT ACTIONS	The statements in this action list are executed instead of the GET ACTIONS for the associated component.

Example

```
REVISION 2.0;

INITIALIZE COMPONENT Reset;

COMPONENT Reset;
    TYPE INTEGER;
    SET ACTIONS;
        POKEINITIAL;
        VALIDATE Short;
        VALIDATE Long;
        SET Xy_is;
    END ACTIONS;
END COMPONENT;                ! Reset

COMPONENT Short;
    TYPE RARRAY 1,10;
END COMPONENT;                ! Short

COMPONENT Long;
    TYPE RARRAY 1,90;
END COMPONENT;                ! Long

COMPONENT Xy_is;
    TYPE DISCRETE;
    VALUES SHORT,LONG;
    INITIAL SHORT;
    SET ACTIONS;
        SELECT Xy_is;
        CASE SHORT;
            HIDE Longp;
            SHOW Shortp;
        CASE LONG;
            HIDE Shortp;
            SHOW Longp;
        END SELECT;
     END ACTIONS;
END COMPONENT;                ! Xy_is
```

(Continued)

```
COMPONENT Acquire;
    TYPE INTEGER;
    GET ACTIONS;
        NOTIFY "Got it!";
    END ACTIONS;
END COMPONENT;                ! Acquire

PANEL Main;
    TOGGLE 100,190,Xy_is;

    PANEL Shortp;
        XY 6,30,Short,1,100,-1,1;
    END PANEL;

    PANEL Longp;
        XY Acquire;
            POSITION 6,30;
            SCALE 1,100,-1,1;
            TRACE Long;
     END TRACE;
        END XY;
    END PANEL;
END PANEL;
```



Index