09/11/90

HP ITG DRIVER WRITER'S MANUAL

The purpose of this document is to provide a guide on how to write a driver for the HP Interactive Test Generator. Background on controlling an instrument and on HP ITG itself is included in chapters I-III for reference and completeness; however, the main focus is on how to write, test and document a driver. Appendices are included on various topics on the operation of HP ITG that are useful to driver writers. It is presumed that the other HP ITG manuals have been read, especially the System Documentation. The main focus is on the organization of HP ITG drivers; for more information on syntax see the HP ITG System Documentation.

I. CONTROLLING AN INSTRUMENT (OVERVIEW)

Any instrument, such as a voltmeter, function generator or oscilloscope, must include some type of interface in order 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 are what allows 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, etc.

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

Primarily for this reason, many instruments provide another type of interface that allows controls to be set by a computer. This type of interface is known as an interface bus. An interface bus will allow many different devices to talk to each other. There are many different protocols that have been defined to allow communication between devices, such as RS-232 or IEEE 488.2 (HP-IB), and 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 devices. As a result, instruments typically speak their own language, which means that a computer must be able to talk to each instrument separately. 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 quickly change the settings of an instrument, much quicker than the front panel interface allows. However, in order to write a program to change these settings, the programmer must have

a knowledge of the language the instrument speaks, that is, the programmer must know what to send to the instrument to change the correct control to the proper setting. This set of commands, known as the command set, have syntax and other rules that the programmer must adhere to. So, by controlling the instrument over the interface bus, settings can be quickly changed and states can be quickly recovered, but writing a program to accomplish these tasks will take longer, in general, than it would to adjust the controls from the front panel.

The HP Interactive Test Generator system combines the speed of the interface bus and the convenience of the front panel interface to provide another type of interface that combines the advantages of the other two. Using HP ITG, the operator no longer needs to be concerned with the command set syntax of the instrument. Instead, HP ITG knows how to communicate with the instruments it is controlling. The information is provided to it in the form of an ASCII file, one for each instrument, called an instrument driver.

II. HP ITG

Purpose

HP Interactive Test Generator is a software development tool that simplifies the task of developing software to control an instrument. It allows the operator to control the instrument with no knowledge of the programming language of the instrument. It also allows the operator to store and recall states of the instrument, and to create programs that control the instrument.

Modes of Operation

There are two main operating environments that the user can operate in: the development environment and the runtime environment. The development environment allows the operator to manually interact with HP ITG by using a mouse to click on controls on the computer screen. It is used to create new states and to generate instrument-controlling program code by changing the settings of the instrument one at a time. The states that are created can be saved by HP ITG, and they can be recalled quickly at any time. The development environment is also known as panelmode operation.

The runtime environment, on the other hand, allows the user to run the program code that was created in the development environment. The runtime environment has a faster execution speed because there is no overhead, such as controlling the panels in the development environment. To a driver writer, the runtime mode of operation is the one that should be focused on. While it is essential to have the driver operate correctly in the development environment, this does not necessarily ensure correct operation in the runtime environment.

Most drivers written for HP ITG do not require that the instrument be present in order for the user to create states. Some drivers that use what is known as a "learn string" do require that an instrument be present, because it is necessary for HP ITG and the instrument to communicate while creating states.

When the instrument is present, the operating environment is "live" and it is known as livemode operation. Without the instrument present, it is referred to as non-live mode operation. While creating states in non-live mode may be more convenient at times, the only way that HP ITG can guarantee a valid state for a learn string driver, (HP ITG knows the instrument's settings correctly), is with livemode operation. Otherwise, HP ITG does not know the current settings of the instrument and may generate invalid code or states. Only drivers that use learn strings have this problem. To find out whether or not a driver uses learn strings for recalling states, consult the help file for a particular driver.

HP ITG Functions

HP ITG has two main functions: to allow the user to be able to quickly create instrument setups, or states, by changing controls on the screen of the computer that correspond to controls on the instrument, and to allow the user to quickly and easily write a computer program that, when it is run, will control the instrument over the instrument's HP-IB interface bus. The user may develop states, generate code, or do both at the same time while in the development environment. These states and/or program code may then be used to control the instrument in the faster runtime environment.

Storing and Recalling States

In order to understand the process of storing and recalling a state, an important distinction must be made. The distinction is between the controls in the instrument and the controls in HP ITG. When the value of a control is changed using ITG, the value is entered into ITG itself. At this point, the instrument has no idea that the value of the control has been changed; it is HP ITG's job to inform the instrument of the change. To inform the instrument of the change, HP ITG uses information stored in the instrument's driver to find the correct command to use, and it sends the command and the new value to the instrument using the HP-IB interface bus discussed in the previous section. As a result, there are actually two values of the same control in existence: one is the setting in the instrument, and the other is what HP ITG has stored in memory. It is HP ITG's job to make sure that these two values always match, ensuring a valid state.

When HP ITG stores a state, it saves the current information on the settings of the controls of the instrument into a file, called a workfile, under a user-selected name. This allows HP ITG to later recall the state into memory, and copy the saved values (the values of the controls when the state was saved) back into the controls. Recalling is usually the fastest way for HP ITG to return itself (and the instrument) to a previous state.

Generating Code

HP ITG can also generate program control code that, when run, will control the instrument in exactly the same manner as happened in the development environment. There are two different ways to log code: as subroutine calls or as basic OUTPUT and ENTER statements. Without going into detail, the

subroutine calls are actually calls to HP ITG subroutines that accomplish the given task. In order for the subroutine calls to work, the HP ITG subroutines must be loaded as part of the program. When "Log Calls" mode is chosen in the HP ITG development environment, the subroutine calls will be logged into the HP ITG editor.

Organization

HP ITG is actually a collection of subroutines that each have a different function. The advantage to the modular design of HP ITG is twofold: it makes it easy for HP ITG to generate subroutine calls that control the instrument, and it makes it possible for the user to quickly and easily generate a program from scratch. A major example of programs that call HP ITG subroutines, aside from HP ITG itself, are the test programs that are used to test a driver. Test programs must have a way to retrieve the current control settings from HP ITG and from the instrument. This is accomplished by calling the appropriate subroutine.

The two most important subroutines in HP ITG are undoubtedly Hpt_set and Hpt_get. These two subroutines basically are what allows HP ITG to talk to the instrument. They use the information in the instrument driver's set and get actions for each component to send HP-IB commands to and from the instrument. When in the development environment, every time the user enters a value into a component or queries a value back from the instrument, either Hpt_set or Hpt_get is called. The code that is generated will also show a call to Hpt_set or Hpt_get.

Instrument drivers are text files that contain instrument-specific information on how the different controls of the instrument interact with each other. Instrument drivers instruct ITG how to communicate with the instrument in a way that the instrument will understand, how the controls behave and interact, and what actions ITG should take to ensure that the list of driver values always match the instrument control values. Along with each driver is a helpfile that ITG uses to generate on-line help for the user.

III. HP ITG DRIVERS

As mentioned above, a driver written for HP ITG is an ASCII file that contains information about the instrument it describes. This section describes in more detail the different parts and functions of drivers.

Organization

All drivers written for HP ITG have three parts: a header, a component section and a panel section. The header section contains general information about the driver. The other two sections describe to HP ITG how to communicate with the instrument and the user, respectively. A minimum driver MUST include:

* A REVISION statement at the top of the driver.

* At least one component declaration.

* A main panel. (Note that panels nest inside one another, so any additional panels must be included as subpanels under the main panel)

The REVISION statement indicates what version of HP ITG the driver was written for. See the documentation to determine the latest revision number to use. The higher the revision number the more features in the ITG driver language you can use. MSO will REQUIRE instrument drivers to be written with the latest revision number possible. As time goes by the driver parsers will become more strict about syntax "errors" that you could get away with with the original products. MSO will strive to strictly enforce the syntax rules in the documentation.

Along with the revision statement at the top of the driver, there can be up to four statements that describe certain components to HP ITG as being special. These are the INITIALIZE COMPONENT, ERROR COMPONENT, STORE COMPONENT and RECALL COMPONENT statements. Each statement controls what ITG does in case of a reset, an error, or when storing or recalling states.

Component Section

The component section of a driver describes the instrument to HP ITG. Each component corresponds to a different control on the instrument, usually on a one-to-one basis. Although the order of the components can make a difference on how the driver behaves (see section on Recalling States), the component section of a driver is not executed by HP ITG in any particular order. Instead, each component is a stand-alone description of a certain part of the instrument that HP ITG uses when the need arises.

All components in a driver will have a name, a type and an initial value. Most components will also have other parts such as set/get actions, panel set/get actions, a coupled statement, or options such as NOTSAVED, NOGEN or NOERRCHECK. In general, each component in a driver will relate to one control on an instrument. The different parts of the component tell HP ITG how to accomplish different tasks that it may need to do.

The set actions of the component tell HP ITG the necessary steps to take to set the value of a control on the instrument to a new value. The get actions give HP ITG the necessary information to ask the instrument what the value of a particular control is, and then read back the data from the instrument in the correct format. Panel set/get actions are exactly the same as set/get actions except they are used only when the driver is being used in the development mode of operation. The panel set/get actions are always executed after the set/get actions, not instead of them.

Couplings inform HP ITG that if it changes the value of this particular control on the instrument, it will need to read the values of other, different controls back from the instrument because they might change. Couplings are executed only in the development environment. In the runtime environment, the couplings are not executed, however, the components named in the coupled statement are marked as invalid, leading to possible complications in state recalls.

During a state recall in the runtime mode, the invalid components are handled in the following way: If the component's value in the stored state was valid, then the new value is copied into the component and the component is added to a list of components that require their set actions to be executed. If the component's value in the stored state was invalid, however, the invalid value in the stored state is copied into the component, but the component will remain invalid in the recalled state. For more information, see the System Documentation, "How Recall Works".

Panel Section

The panel sections of the driver tells HP ITG how to create the user interface. Usually, the panel section is easy to write and it follows the menu structure of the instrument's HP-IB command set. That is, the panel section will include a sub-panel for every different menu path on the HP-IB command set. Although it is possible to write a driver that does not reflect the HP-IB command set menu structure, there are several good reasons to follow this approach:

1. The overhead is much less. When a driver is written with a different menu structure than the instrument has, extra code is almost always involved.

2. For users of HP ITG that are already familiar with the HP-IB operation of the instrument, the panels will be easier to use.

3. 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 driver will end up appearing and behaving similarly to the front panel of the instrument.

4. Instrument menu structures are not developed at random. They are developed to fit the functionality of the instrument. Writing a driver 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 driver. Each panel contains one or more "widgets", or controls, that correspond to components in the driver. The type of each controls is associated with the type of each component in the driver.

IV. WRITING A DRIVER

Timeline in Writing a Driver

Writing a driver is by no means a simple task. The time in writing, testing and documenting a good driver is directly proportional to the complexity of the instrument:

Simple instrument : 2-4 weeks (HP3437, HP5384 drivers)

Moderate complexity instrument : ~2 months (HP3314, HP3458 drivers)

Complex instrument : ~3 months (HP54501, HP71000 drivers)

Writing a driver for the first time is even more difficult because not only is the instrument new, but HP ITG and the driver instrument descriptor language is also new. However, the design phase is the most important phase in the development of a driver. If the design phase is done well, it is fairly straightforward to code the driver.

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

instrument control - an instrument control is a control on the instrument that affects the operation of the instrument. 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 necessarily all the time, 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 a driver 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 driver, because the HP-IB command set is the way HP ITG communicates with the instrument. In addition, the front panel operation of the instrument can give a good start on laying out the panel section of the driver.

One good way to learn the instrument's HP-IB programming language is to write BASIC programs that do certain tasks that the instrument can be used for. An example, for an oscilloscope, is using the autoscale feature to set up a trace on the scope screen and asking for measurements, etc. The advantage of learning the instrument in this way is that not only will you learn the details of the HP-IB command set, but once the driver is done you can go back and use HP ITG to write the same programs. The code HP ITG generates should be comparable to the code you originally wrote, so this provides an easy way to

measure the effeciency of the driver later.

Pay special attention to the details while learning the instrument's operation, because it is 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 bus, and some HP-IB features may not reside on the front panel anywhere. The function of some of the commands over the bus may or may not be the same as the same features on the front panel. For instruments with a menu-driven front panel and/or softkeys, does the HP-IB command set reflect the same menu structure? Are some commands valid at some times but will not work at other times? All of these questions and more need to be thought out before writing the driver in order to minimize the time in driver coding and testing.

The following outline summarizes the procedure that should be followed to learn the instrument, in order to effectively anticipate driver design problems:

I. Identify the controls of the instrument

This is usually an easy task to do from the front panel, however, sometimes it can get tricky matching HP-IB commands to instrument controls. The idea is to assign the controls of the instrument (frequency controls, voltage controls, etc.) to HP-IB commands and queries. Controls for which there is no HP-IB command or query are therefore impossible to implement into a driver, because there is no way for HP ITG 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 driver 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.

It is not always obvious from perusing the instrument's command set, which commmands relate to separate controls on the instrument. For example, 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.

II. Identify the menu structure of the instrument, if there is any.

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. An example of a menu structure from the HP 54111D

digitizing oscilloscope is:

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, so the command will work no matter when it 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.

III. Isolate the various modes of the instrument.

An instrument mode is a state or a set of states of the instrument that requires an additional set of controls, valid only for that state, to uniquely determine the state of the instrument. 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 edge of the signal or the 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, say when channel 1 is high and channel 2 is low. The current setting of the edge control makes no difference at all in the operation of the oscilloscope when it is set in 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 will 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 effect, 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.

Another way to look at the difference between instrument menus and instrument modes is the following: instrument menus have to do with when HP-IB commands are valid or not, while instrument modes indicate if 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 be not 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 driver writer. The controls under a certain mode are not valid in other modes, so in order 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, in the runtime environment the user could unknowingly change the mode of operation of the instrument by changing a control under the mode. The driver needs to take this into account. See the section on Developing Set and Get actions.

IV. 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.

V. Identify the range or values of each control

Every control on the instrument should have an absolute maximum and an absolute minimum value. These values must be found before writing the driver to cut down on much hassle later. Be aware that sometimes the maximum and minimum values of a control will change depending on the settings of other controls on the instrument. An example of this, from the HP 54111D driver is the probe adjustment control. For the initial setting, 1:1 probe adjustment, the channel sensitivity will have a range of 1mV to 5V. With a probe adjustment of 10:1, however, the sensitivity will range from 10mV to 50V. Therefore, the range of values for channel sensitivity depends on the setting of the probe adjustment control. The driver was written to accomodate the absolute minimum and maximum values for the channel sensitivity control.

It is possible, however, to use "soft ranges" in the driver so that the correct maximum and minimum values are reflected in the driver at all times. To do this, the driver writer must spend some time trying to figure out the algorithm that is used by the instrument to come up with 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 driver can be a complex and sometimes can lead to conflicts with other components due to couplings and GETs that must be added. The use of soft ranges is therefore time-consuming and adds extra code to the driver. For complex drivers, then, it is probably better to use absolute limits with verification in the panel set actions.

For discrete controls, there will be a set of choices that the control can be

set to, instead of a range of choices. In some cases, the available discrete choices can change depending on the settings of other components in the driver. 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 on/off discrete options at appropriate times.

VI. 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 which will "survive" an instrument reset and turning the instrument off. Such controls present a special headache to driver writers if the problem is 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 driver.

VII. Summarize the HP-IB command set

Every control that you decide to include in the driver 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 example entry is for an oscilloscope)

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 driver are usually developed in parallel; that is, as another component is added, the panel widget that goes with that component is added at the same time.

It helps while writing the driver code to periodically check the code with the hpidc parser. This is a HP-UX tool that goes through the driver and locates errors much faster than the HP ITG parser does. For more information on hpidc, see Appendix B.

A. Prepare a skeleton component outline

Recall that every component in the driver must have a name, a type and should have an initial value. Using the table that was prepared above, by assigning exactly one component to each control, a skeleton driver can be written quickly from the information in the table. Usually it is helpful to name each component something similar to the control it represents, and to group

FINISH

components in the same menus and modes together. The driver 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 the values statement will have a different syntax depending on the type of the component. See the HP ITG System Documentation for details.

B. 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:

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 they 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;
END COMPONENT;

COMPONENT DUMMY_COMPONENT;
  TYPE INTEGER;
  INITIAL INVALID;
  SET ACTIONS;
    GOSUB Get_data_from_instrument;
  END ACTIONS;
END COMPONENT;
```

Then, the DUMMY_COMPONENT can be assigned to a button on the panel of the driver. When the user clicks on the panel button, HP ITG will execute the set actions of the DUMMY_COMPONENT, which in turn will get the array from the instrument.

The panel section of the driver, as explained in the System Documentation, consists of panels, nested inside one another. Each widget that is added to the driver will have to be inside one of these panels. An easy way to design the panel structure of the driver 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 driver, 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.

Note about adding subpanels: Once you have subpanels in the driver you must add code in the RESET COMPONENT to SHOW or HIDE the appropriate panels. Only the root panel is automatically shown by HP ITG. The use of a menu component with SHOWs and HIDEs can be useful, also, as shown below for the HP 54111D driver:

```
COMPONENT RESET NOTSAVED NOGEN;
  TYPE INTEGER;
  INITIAL DONTCARE;
  SET ACTIONS;
    CLEAR;
    OUTPUT STRING "RESET;";
    OUTPUT STRING "HEAD OFF; LONG OFF;";
    FLUSH;
    POKEINITIAL;
    GOSUB RESET_ACTIONS;
    IF PANELMODE THEN;
      SHOW ROOT,CRT;
      SHOW ROOT,CONTROLS;
      GOSUB UPDATE_TRACE_DATA;
    END IF;
  END ACTIONS;
END COMPONENT;

    DISCRETE MAIN_MENU;
      POSITION 65,188;
      SIZE 143,20;
      BACKGROUND (75,0,240),0;
      LABEL "Channel 1 Panel","Channel 2 Panel","Timebase Panel",
          "Trigger Panel","Display Panel","Delta V Panel",
          "Delta T Panel","Memory Panel","Function Panel",
          "Measure Panel","Utility Panel","Status Panel",
          "About Panel";
      UPDATE ACTIONS;
        HIDE CONTROLS,CHANNEL1_SUBSYSTEM;
        HIDE CONTROLS,CHANNEL2_SUBSYSTEM;
        HIDE CONTROLS,TIMEBASE_SUBSYSTEM;
        HIDE CONTROLS,TRIGGER_SUBSYSTEM;
        HIDE CONTROLS,DISPLAY_SUBSYSTEM;
        HIDE CONTROLS,VMARKERS_SUBSYSTEM;
        HIDE CONTROLS,TMARKERS_SUBSYSTEM;
        HIDE CONTROLS,WAVESAVE1_SUBSYSTEM;
        HIDE CONTROLS,WAVESAVE2_SUBSYSTEM;
        HIDE CONTROLS,WAVEMATH_SUBSYSTEM;
        HIDE CONTROLS,MEASURE_SUBSYSTEM;
        HIDE CONTROLS,UTILITY_SUBSYSTEM;
        HIDE CONTROLS,STATUS_PANEL;
        HIDE CONTROLS,ABOUT_PANEL;

        SELECT MAIN_MENU;
        CASE CHAN1;
```

```
      SHOW CONTROLS,CHANNEL1_SUBSYSTEM;
    CASE CHAN2;
      SHOW CONTROLS,CHANNEL2_SUBSYSTEM;
    CASE TIME;
      SHOW CONTROLS,TIMEBASE_SUBSYSTEM;
    CASE TRIG;
      SHOW CONTROLS,TRIGGER_SUBSYSTEM;
    CASE DISP;
      SHOW CONTROLS,DISPLAY_SUBSYSTEM;
    CASE DELTV;
      SHOW CONTROLS,VMARKERS_SUBSYSTEM;
    CASE DELTT;
      SHOW CONTROLS,TMARKERS_SUBSYSTEM;
    CASE WAVS;
      SELECT DISPLAY_MODE;
      CASE REPETITIVE;
        SHOW CONTROLS,WAVESAVE1_SUBSYSTEM;
      CASE REALTIME;
        SHOW CONTROLS,WAVESAVE2_SUBSYSTEM;
      END SELECT;
    CASE WAVM;
      SHOW CONTROLS,WAVEMATH_SUBSYSTEM;
    CASE MEAS;
      SHOW CONTROLS,MEASURE_SUBSYSTEM;
    CASE UTIL;
      SHOW CONTROLS,UTILITY_SUBSYSTEM;
    CASE STATUS;
      SHOW CONTROLS,STATUS_PANEL;
    CASE ABOUT;
      SHOW CONTROLS,ABOUT_PANEL;
    END SELECT;
  END ACTIONS;
END DISCRETE;
```

The main menu component uses UPDATE ACTIONS as the means for accomplishing the appropriate SHOWs and HIDEs every time the component is updated, in panelmode only. To allow the driver to work properly on HP ITG/DOS, the action list should first HIDE all the subpanels and then SHOW the appropriate one. If this method is not used, the panels under HP ITG/DOS will flash every time the main menu component is changed.

C. Developing the set and get actions

For every component in an HP ITG driver 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 driver or to do 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 will accomplish this using the OUTPUT and ENTER statements, which are covered in the System Documentation. 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 get actions above use a dummy component, charstring, to read a string from the instrument, and then they compare charstring to the different possible instrument responses to fetch the correct discrete value for Component_name onto the stack. The value is then stored into the component by the STORE statement. An important thing to remember when using string comparisons with ITG is that 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 A's 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 128;
  INITIAL INVALID;
END COMPONENT;
```

A string length of 128 is usually long enough to read most strings that may come back from the instrument.

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, HP ITG will issue a NO CASE(x) 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. The problem with instrument modes is the following. Suppose the instrument is currently operating in a certain mode, and you want to set or get a component/control in a different mode. You must therefore include, as part of the HP-IB command string that the component sends out, a prefix that changes the mode of the instrument to the correct one 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 simply output an HP-IB command that changes the mode back to the old one, because you don't know what the old mode of operation was. The solution is to execute the set actions of the mode component that contains the needed information of what the old mode of operation was. This will leave everyone happy: HP ITG has a correct state, the component has been set/queried, and the instrument is still in the old mode of operation. An example of this from the HP 54111D driver 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 old mode
  END ACTIONS;
  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. EDGE_TRIGGER_SOURCE is a component that 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 current menu to trigger and the current mode to edge on the instrument. 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 old mode of operation. Note that this method will work no matter what the old mode of operation was, but if the old 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 shown above. Some instruments, such as the HP 54111D have modes that are nested two deep. For example, in the HP 54111D, there is a different trigger level control depending on which source is selected. Therefore, to set/query the trigger_level_ch1 component, first the edge mode of operation is selected and then the edge trigger source is set to channel 1, before the trigger_level_ch1 control is even 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, which sets the appropriate menus/modes. To restore the old modes, the set actions for edge_trigger_source are executed, which in turn will execute the set actions for trigger_mode, as discussed above.

This completes the basic driver. Undoubtedly, there will be situations that will call for some creativity that no manual can cover effectively. The best way to deal with such problems is to experiment with different solutions and choose the best one. For examples of some programming tricks, or other hints in programming the set/get actions, see appendix C.

While it is possible to add all the components at once, it is advisable to add them one at a time, and read the unfinished driver into HP ITG to check whether the code actually works. This will save time in debugging a huge driver at a later time. You can also use the HP Instrument Driver Compiler (hpidc) to check for syntax or spelling errors in the driver. Hpidc is much faster than HP ITG's parsing routines, but you can't check the actual operation of the code with hpidc. MSO recommends that the hpidc program be used to check for

any syntax errors, but the semantic/logic testing must still be done in other ways. For more information see the appendix on hpidc.

D. Finishing touches

Couplings

Couplings are situations in the instrument that occur when by changing the value of a particular control on the instrument, another control on the instrument might be affected. To keep HP ITG up-to-date on the current state of the instrument, these coupling situations must be covered in the driver. This is done by using the coupled statement. At this point in the development process, it is OK to add the obvious couplings that you know about to the driver, but there is no need to try and find all the hidden couplings that are bound to be in the instrument. These hidden couplings will be easily uncovered in the testing phase of driver development. See the section on Interpreting Test Results in the chapter on Testing the driver.

Panel set and get actions

Sometimes there is a situation in the driver that things should happen in the development environment to keep the panels working correctly, but they are unnecessary in the runtime environment. To make the driver operate as fast as possible in the runtime environment, any such overhead should be placed in the panel set/get actions, rather than the set/get actions. HP ITG will then know to skip the unnecessary actions in the runtime environment.

Dummy components

Components in a driver 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. This is a perfectly legal thing to do, and other components in the driver can then reference the value in the component. This can be used as a method of parameter passing between two components, or other functions such as remembering the last value of components or simply as a placeholder in a SELECT..CASE..ELSE construct, which requires that a component name be specified. Since these components are only used as variables, always use the NOTSAVED NOGEN NOERRCHECK options with them to save time and space in the driver.

Valid Reset State

A working driver must have a valid state at all times (that is, that HP ITG's values in the components match the instrument's control settings). To do this, HP ITG tracks the state of the instrument using the driver as a guide of how to do it. There needs to be a state from which it is guaranteed that the state of the driver and the state of the instrument are exactly the same. This is known as a reset state.

The reset component, specified by the INITIALIZE COMPONENT statement in the

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

When one or more components/controls are known to survive a reset, the initialize component should GET the appropriate components as part of its set actions. 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 mess up the values of other controls of the instrument.

2. Using a get preserves the functionality of the instrument. That is, after a reset, the driver will reflect a natural state of the instrument, rather than vice-versa. The entire purpose of the driver is to allow HP ITG 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 this is the case, the error component can be used for automatic error checking in the driver. The ERROR COMPONENT statement at the top of the driver declares which component in the driver is the error component.

When the error checking mode in HP ITG is turned on, each time there is a set or a get on any component, HP ITG will execute the set actions of the error component, unless the NOERRCHECK option is set (see NOTSAVED, NOGEN, NOERRCHECK) for that component or SKIPERRORCHECK is done in the action list. If, after the set actions have been executed, the error component's value is zero, HP ITG will take no other action. Otherwise, the component's number is taken as the error number and HP ITG 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. An example of this, taken from the HP 54501A driver, is included below:

```
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 particular instrument, if a zero is returned, there has been no error and the set 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. The actions of flush_err_queue are an example of recursion in an HP ITG driver and are included in the appendix on programming practices.

NOTSAVED, NOGEN, NOERRCHECK

The notsaved, nogen and noerrcheck flags in component definitions are used to signal HP ITG when a particular component is special. They are used as follows:

NOTSAVED - used when a particular component does not contain a value representing a control on the instrument. Typical types of components that notsaved is used on are dummy components that contain a value but do not relate to any setting in the instrument, menu components that do not have any impact on the operation of the instrument, and action components that implement some automatic setup feature of the instrument into the driver, but do not contain any value, and components that hold data returned from the instrument.

NOGEN - when HP ITG generates code, it does so whenever a component is accessed. The nogen flag tells HP ITG not to generate code for this particular component. Menu components and dummy components that relate to operation in panelsmode should generally be specified as nogen components.

NOERRCHECK - HP ITG automatically checks for errors every time there is an interaction between the instrument and the driver in a component's set or get actions. This is sometimes unnecessary and to save time, some components can be marked with the noerrcheck flag. Components such as the menu component that do not interact with the instrument are usually NOERRCHECK.

Recalling States

HP ITG was meant to work properly in this category without any modification to the driver. However it is always necessary to make some changes to the driver in practice, due to problems that are usually uncovered in testing the driver. See the section on "Recall testing" in the section on testing the driver.

V. Testing the Driver

Testing a driver can take as much time as writing the code itself. There are three ways to test a driver: panelmode tests, recall tests (regression tests), and manual testing. All three methods of testing use two programs called test_help and test_gen to generate the program code that tests the driver and compares the driver values to the instrument values. Manual testing can only be used with HP ITG/RMB, not with HP ITG/DOS or Short Circuit.

Panelmode tests are used to verify that the driver's state will match the state of the instrument in the panels mode. Similarly, recall testing verifies the way the driver recalls states. Recall testing usually is run in the runtime environment, although it does not have to. Recall tests will take over an hour to run, and can take days or even weeks to complete. It is often necessary to limit the number of states to make the recall tests run faster. See the section on recall testing in this section for more information. Manual testing can be a valuable way to check the obvious couplings and components, because the driver writer will usually know the parts of the driver that may not be fully debugged.

In this section, all three methods of testing will be covered, and ways to debug and interpret test results will be covered last. For more complex instruments, manual testing is impractical for uncovering every flaw in the driver, but it can still be a valuable tool in gaining confidence in the panelmode operation of the driver. The methods of automatic testing described in this section use both the test_help and test_gen awk scripts, which operate in the HP-UX operating environment. These methods of testing will not work in a workstation system, though the HP Basic programs, once they are generated, can be used on a workstation system.

Testing can be started before the driver is completed, but be aware that if any new components are added the test programs will have to be modified or regenerated from scratch. This can be a very time consuming process, because for complex instruments test_gen can take over an hour to run. With this in mind, the best method for testing would probably be to test infrequently while still in the development stage.

Running a test consists of two stages: generating the test code and running the test. The debugging stage comes later after the test has been run. Changes in the driver can sometimes result in an obsolete test program. To know when a test program becomes obsolete, the following guidelines can be used:

Your test program will need to be modified, or re-generated if:

- You add any new components to the driver, or delete any old ones.

- You change a component's name.

- You change the values or values range of any component that has a test...endtest entry in the test code.

For simple changes to the driver, it may be easier to edit the test code directly. Also note that new test programs will not work with old workfiles,

in general.

BEFORE GENERATING NEW TEST CODE, RE-STORE THE DRIVER IN AN UP-TO-DATE WORKFILE!

Using test_help and test_gen

Generating test program code consists of two steps:

1. Running test_help and modifying the output to specify the test that you want to run, the components that are to be tested, and the values of the components that should be tested.

2. Running test_gen to convert the file you created in step 1 into a working test program that can be run with little or no modifications. The test program will be in HP Basic, and it will use HP ITG subroutines to do its testing.

The syntax for running test_help is:

```
test_help ID_file [test_output_file]
```

where ID_file is the filename of your driver, and the optional test_output_file is the filename that you want test_help to put its output into. Once test_help has been run, the output file will consist of a series of test descriptions. For the following driver, the test_help output will be as shown below the driver:

```
REVISION 1.1;

COMPONENT INTEGER_COMPONENT;
  TYPE INTEGER;
  VALUES RANGE 0,100;
END COMPONENT;

COMPONENT REAL_COMPONENT;
  TYPE CONTINUOUS;
  VALUES RANGE 0,1E6;
END COMPONENT;

COMPONENT DISCRETE_COMPONENT;
  TYPE DISCRETE;
  VALUES LOW,MID,HIGH;
END COMPONENT;

COMPONENT DUMMY COMPONENT;
  TYPE INTEGER;
END COMPONENT;

PANEL ROOT;
END PANEL;
```

Test_help output:

```
OPTIONS USEGETS TRACE
TEST INTEGER_COMPONENT
  SET INTEGER_COMPONENT 0
  COMPARE
  SET INTEGER_COMPONENT 50
  COMPARE
  SET INTEGER_COMPONENT 100
  COMPARE
ENDTEST
!
TEST REAL_COMPONENT
  SET REAL_COMPONENT 0
  COMPARE
  SET REAL_COMPONENT 500000
  COMPARE
  SET REAL_COMPONENT 1000000
  COMPARE
ENDTEST
!
TEST DISCRETE_COMPONENT
  SET DISCRETE_COMPONENT LOW
  COMPARE
  SET DISCRETE_COMPONENT MID
  COMPARE
  SET DISCRETE_COMPONENT HIGH
  COMPARE
ENDTEST
!
TEST DUMMY_COMPONENT
  ! Warning: no values range for this component
  SET DUMMY_COMPONENT -1
  COMPARE
  SET DUMMY_COMPONENT 0
  COMPARE
  SET DUMMY_COMPONENT 1
  COMPARE
ENDTEST
!
```

The test_help output consists of one test description entry for each component in the driver. Each test entry sets the respective component to its lowest, middle and high value for real/integer components. For discrete components, there is one set for each discrete value. When test_help can't find a value range for a component the default values are -1, 0 and 1. The test_help output is in a form that can be read by test_gen directly, but you will want to modify the test entries first, before running test_gen, to customize the test to what you want to run.

Each test entry describes the test that will be run on this component. For example, the test entry for discrete_component, above, tells test_gen that you want to set the component to low, then compare the values of every component in the driver to their corresponding controls on the instrument, etc. The COMPARE keyword means that EVERY component will be compared to its corresponding

instrument control, not just the component being tested. In this way, coupling problems or other component interactions can be quickly discovered.

When you edit the test_help output, you will want to customize the test entries to streamline the tests. For example, it makes no sense to test a dummy component such as charstring in the previous section, because this component does not correspond to a control on the instrument. To change the test description file to reflect this, the test entry for the dummy component is removed and replaced with an instruction to ignore the component. Removing the test entry from the file keeps a test from being run on this component, and the ignore statement tells test_gen to exclude the component from being compared when the COMPARE keyword is encountered. By removing the dummy component from the test program, the result is that the test will run faster.

The test entry:

```
TEST DUMMY_COMPONENT
  ! Warning: no values range for this component
  SET DUMMY_COMPONENT -1
  COMPARE
  SET DUMMY_COMPONENT 0
  COMPARE
  SET DUMMY_COMPONENT 1
  COMPARE
ENDTEST
```

is replaced with:

```
IGNORE DUMMY_COMPONENT
```

Test_gen

Test_gen is the program that turns the test description file generated by test_help into a ready-to-run basic program. To do this, it generates a main program that has a basic variable for every component in the driver. This main program has a section devoted to each test entry that exists in the test description file. For the test description file above, there will be three sections in the test_gen main program.

The syntax for running test_gen is:

```
test_gen test_description_file ID_file device_name [RMB output]
```

where test_description_file is the edited version of test_help's output, ID_file is the filename of the driver, device is the name of the instrument that the driver was written for, and RMB output is an optional filename for test_gen's output.

IMPORTANT: test_gen uses the device_name parameter to generate a subroutine call to an HP ITG subroutine that initializes the driver and the workfile. For this reason, device_name should match the name of the instrument contained in the workfile.

When test_gen encounters the COMPARE keyword, it generates a subroutine call in the basic program to a very useful subroutine called tester_compare. Tester_ compare, also generated by test_gen, checks to see if the current state of the driver matches the current state of the instrument. It accomplishes this in three steps. First, it copies the values of every component being compared to a basic variable. Second, it calls another subroutine called read_instrument, that reads the values of all the controls in the instrument into their respective components. Last, it compares the new values of all the components with their old values, before read_instrument was called. If the two values do not match for a component, a message is generated to inform the user of the mismatch. The message will have the following form:

Test <Component_name> component TIME_SENS: Instr was <xxx>, ITG expected <yyy>

where xxx is the actual value of the control in the instrument, and yyy is the value of the component in the driver.

Test_gen keywords

The options statement at the top of the test_help output tells test_gen the category of test program that should be generated. The four parameters that can be used with the options statement are described below:

USEGETS - Signifies to test_gen that you want it to generate the subroutine Read_instrument. Otherwise, you can write your own Read_instrument subroutine. Writing your own is rather involved, so it is better to let test_gen generate it for you.

RECALL - Instructs test_gen to generate code for recall testing. See the section on recall testing.

TRACE - Test_gen will add print statements at several points in the test program to allow you to follow the progress of the test.

RECHECK - This parameter, valid only for recall tests, tests for a certain type of error that can occur when recalling states. See the section on recall testing.

TEST...ENDTEST describes a test of a certain component. The code inside the test...endtest statement consists of sets with compares after each set.

IGNORE instructs test_gen to leave the specified component out of the tester_compare subroutine. That is, the tester_compare subroutine will not compare the component when it is called. In general, any component with only get actions and no set actions (with the notable exception of the error component) should be ignored.

Other keywords that are not used as often are described in the appendix on test_help and test_gen.

Example test_gen output:

The test_gen output corresponding to the test description file, above, with the dummy_component test entry replaced with an ignore statement, is included below. The code will be different for recall tests, but the form will be essentially the same:

```
1 ! Generated program to test ID for HPdevice_name
2 OPTION BASE 1
3 COM /Tester/ INTEGER Test_device,Test_name$[256]
4 COM /Tester/ INTEGER Iparm,Test_last,Test_recall
5 COM /Inst_state/ INTEGER H_Integer_comp
6 COM /Inst_state/ INTEGER H_Real_compone
7 COM /Inst_state/ INTEGER H_Discrete_com
8 COM /Inst_state/ INTEGER H_Dummy_compon
9 !
10 REAL Integer_comp
11 REAL Real_compone
12 DIM Discrete_com$[25]
13 REAL Dummy_compon
14 !
15 Hpt_dvrcom
16 Hpt_init("WFDFLT")
17 Hpt_assign(Test_device,"HPdevice_name")
18 Hpt_monitor(Test_device,1)
19 Test_last=0
20 Test_recall=0
21 !
22 Hpt_assigncomp(H_Integer_comp,Test_device,"INTEGER_COMPONENT")
23 Hpt_assigncomp(H_Real_compone,Test_device,"REAL_COMPONENT")
24 Hpt_assigncomp(H_Discrete_com,Test_device,"DISCRETE_COMPONENT")
25 Hpt_assigncomp(H_Dummy_compon,Test_device,"DUMMY_COMPONENT")
26 !
27 Integer_comp: Test_name$="INTEGER_COMPONENT"
28   PRINT "Trace: test INTEGER_COMPONENT"
29   Hpt_set2(H_Integer_comp,0)
30   Tester_compare
31   Hpt_set2(H_Integer_comp,50)
32   Tester_compare
33   Hpt_set2(H_Integer_comp,100)
34   Tester_compare
35   !
36 Real_compone: Test_name$="REAL_COMPONENT"
37   PRINT "Trace: test REAL_COMPONENT"
38   Hpt_set2(H_Real_compone,0)
39   Tester_compare
40   Hpt_set2(H_Real_compone,500000)
41   Tester_compare
42   Hpt_set2(H_Real_compone,1000000)
43   Tester_compare
44   !
45 Discrete_com: Test_name$="DISCRETE_COMPONENT"
46   PRINT "Trace: test DISCRETE_COMPONENT"
47   Hpt_set_str2(H_Discrete_com,"LOW")
48   Tester_compare
```

```
49   Hpt_set_str2(H_Discrete_com,"MID")
50   Tester_compare
51   Hpt_set_str2(H_Discrete_com,"HIGH")
52   Tester_compare
53   !
54   !
55 END
56 !
57 SUB Tester_compare
58   OPTION BASE 1
59 COM /Tester/ INTEGER Test_device,Test_name$
60 COM /Tester/ INTEGER Iparm,Test_last,Test_recall
61   COM /Inst_state/ INTEGER H_Integer_comp
62   COM /Inst_state/ INTEGER H_Real_compone
63   COM /Inst_state/ INTEGER H_Discrete_com
64   COM /Inst_state/ INTEGER H_Dummy_compon
65   REAL Dummy,Dummy_r(1:1,1:5000),Dummy_r_c(1:1,1:5000)
66   INTEGER Dummy_i(1:1,1:5000),Dummy_i_c(1:1,1:5000)
67   DIM Dummy$[25]
68   REAL Integer_comp
69   INTEGER S_Integer_comp
70   REAL Real_compone
71   INTEGER S_Real_compone
72   DIM Discrete_com$[25]
73   INTEGER S_Discrete_com
74   Tester_compare: !
75   PRINT "  Trace: in Tester_compare"
76   Hpt_monitor(Test_device,0)
77   Hpt_compaccess(H_Integer_comp,"state",2,S_Integer_comp)
78   Hpt_peek2(H_Integer_comp,Integer_comp)
79   Hpt_compaccess(H_Real_compone,"state",2,S_Real_compone)
80   Hpt_peek2(H_Real_compone,Real_compone)
81   Hpt_compaccess(H_Discrete_com,"state",2,S_Discrete_com)
82   Hpt_peek_str2(H_Discrete_com,Discrete_com$)
83   !
84   Read_instrument
85   !
86   Hpt_peek2(H_Integer_comp,Dummy)
87   IF ((S_Integer_comp=1) AND (Integer_comp<>Dummy)) THEN CALL
Tester_error("INTEGER_COMPONENT",VAL$(Dummy),VAL$(Integer_comp))
88   Hpt_peek2(H_Real_compone,Dummy)
89   IF ((S_Real_compone=1) AND (Real_compone<>Dummy)) THEN CALL
Tester_error("REAL_COMPONENT",VAL$(Dummy),VAL$(Real_compone))
90   Hpt_peek_str2(H_Discrete_com,Dummy$)
91   IF ((S_Discrete_com=1) AND (Discrete_com$<>Dummy$)) THEN CALL
Tester_error("DISCRETE_COMPONENT",Dummy$,Discrete_com$)
92   Hpt_monitor(Test_device,1)
93 SUBEND
94 !
95 SUB Read_instrument
96   OPTION BASE 1
97 COM /Tester/ INTEGER Test_device,Test_name$
98 COM /Tester/ INTEGER Iparm,Test_last,Test_recall
99   COM /Inst_state/ INTEGER H_Integer_comp
```

```
100   COM /Inst_state/ INTEGER H_Real_compone
101   COM /Inst_state/ INTEGER H_Discrete_com
102   COM /Inst_state/ INTEGER H_Dummy_compon
103   REAL Integer_comp
104   REAL Real_compone
105   DIM Discrete_com$[25]
106   REAL Dummy_compon
107   Read_instrument: !
108   ! LOOK OVER AND/OR ADD YOUR OWN CODE !
109   Hpt_get2(H_Integer_comp,Integer_comp)
110  !Hpt_poke2(H_Integer_comp,Integer_comp)
111  !Hpt_setstate2(H_Integer_comp,1)
112   Hpt_get2(H_Real_compone,Real_compone)
113  !Hpt_poke2(H_Real_compone,Real_compone)
114  !Hpt_setstate2(H_Real_compone,1)
115   Hpt_get_str2(H_Discrete_com,Discrete_com$)
116  !Hpt_poke_str2(H_Discrete_com,Discrete_com$)
117  !Hpt_setstate2(H_Discrete_com,1)
118 SUBEND
119 !
120 SUB Tester_error(Name$,Exp_value$,Act_value$)
121 COM /Tester/ INTEGER Test_device,Test_name$
122 COM /Tester/ INTEGER Iparm,Test_last,Test_recall
123   Tester_error: !
124   PRINT "Test ";Test_name$;" component ";Name$;
125   PRINT ": Instr was ";Act_value$;", ITG expected ";Exp_value$
126   PRINT
127 SUBEND
128 !
```

For more information on test_help or test_gen, see the Appendix containing the manual pages on test_help and test_gen at the end of this manual.

PANELMODE TESTING

The first step in running a panelmode test is to run test_gen as described above to generate the test program you want. The result will be an RMB program in ASCII format. To generate the final, working panelmode test program, the following steps need to be done from the rmb environment:

1. GET "<ascii test filename>"

2. Change the line:

   ```
   Hpt_init("WFDFLT")
   ```

   to:

   ```
   Hpt_init("WFDFLT",1)
   ```

3. Delete all subroutine calls to Hpt_monitor. Type:

   ```
   FIND "Hpt_monitor"
   ```

   and press the delete line key until no more lines are found.

   NOTE: these subroutine calls can be left in if you change the call in Hpt_edexins so that nothing is put into the HP ITG editor and

is simply PRINTed to the screen.

4. LOADSUB ALL FROM "HPITG"

5. EDIT Hpt_dlgerror

6. Delete the line, BEEP, and the next line, from the subroutine Hpt_dlgerror.

7. In place of the two deleted lines, add the line
   PRINT Text$(*)

The test program is now complete, so save it on disk so you can run it easily in the future. The program will output a bunch of print statements, so it will be necessary to assign the output to a text file with the PRINTER IS statement. The program can now be run in the same way you would run any basic program, provided that the workfile, WFDFLT, is up to date with the proper driver and instrument name.

RECALL TESTING

Recall testing, or regression testing, tests the driver's ability to accurately store and recall valid states. Recall testing consists of two parts, creating the states and then recalling every state that has been created from every other state. For longer drivers, the amount of time a recall test can take to run can easily be days. For this reason, the amount of states generated in the first part of the test should be limited to no more than 150 states. For example, the HP 54111D driver took approximately 10 seconds to test per recall, and by limiting the test to 80 states the test took only 1 week to run. If the full set of states proposed by test_help had been used, over 500 states would have been generated and the test would have ran for over a year.

The length of time that a test program can take to run depends on:

1) Hardware
2) RMB/WS vs. RMB/UX
3) Complexity of the instrument

The HP 3314 driver test included 120 states and ran for approximately 8-20 hours depending on the hardware. On the other hand, the HP 8508 test with 60 states took 1-2 hours.

The bottom line is for the driver writer to realize that recall testing is a n factorial type of program and deleting a few states can halve the test time. Recall tests should therefore include as few states as can possibly be done.

During the creating states phase, the test program runs in the much the same way that it did during panelmode testing, above. After a component is changed to a new value, tester_compare is called and then the state is saved. After the last component has been changed, the test program begins recall testing in which every state will be recalled from every other state. For short drivers, the original test_help file can be used to generated the recall test program, with the options recall keyword added at the front.

For longer drivers, the test_help output will need to be cut down to limit the amount of states. Usually it is best to leave only one state per component. Any components that are working reliably can be removed from the test, but they should not be IGNOREd to ensure that couplings will be found. It will be impossible to test every possible state in the driver, so it is the writer's choice of which states will be tested, and which tests will be removed to shorten the test time.

The example test_help output, above, modified to only one state per component, is shown below, as a possible example of a recall test description:

```
OPTIONS USEGETS TRACE RECALL
TEST INTEGER_COMPONENT
  SET INTEGER_COMPONENT 50
  COMPARE
ENDTEST
!
TEST REAL_COMPONENT
  SET REAL_COMPONENT 1000000
  COMPARE
ENDTEST
!
TEST DISCRETE_COMPONENT
  SET DISCRETE_COMPONENT LOW
  COMPARE
ENDTEST
!
IGNORE DUMMY_COMPONENT
!
```

Notice that the RECALL keyword has been added to the OPTIONS statement at the top, and each test description entry has only one SET..COMPARE sequence. The choice of the value to set each component to is arbitrary, but it works best if the component is set to something other than its initial value. This example file, when run as a recall test, will generate three states, with one being created each time the routine tester_compare is called. The original test_help output would have resulted in 9 states, or 12 states if the dummy component's test entry is not replaced with an ignore.

The RECHECK keyword can be optionally added to the OPTIONS statement at the top of the test_help file, also. This keyword signals to test_gen to generate code that will test the values that actually end up in the driver to the values that were originally in the state. This amounts to making sure that the driver is working correctly, as opposed to the instrument not reaching the correct configuration, which is the other error that can happen during a state recall. For more information on RECHECK, see the appendix containing the man page for test_gen at the end of this document.

To create the test program, the modified test_help file must be run through test_gen, and the procedures described above in the section on panelmode testing are followed.

Getting State Recalls to Work

Without Learn string...

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

Obviously, 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, there would be no problems with recall testing (at least in theory) other than couplings.

There is a way to get around most problems that have to do with recalling states. This involves the use of a recall component and a finish recall component. The recall component is a special component that HP ITG executes the set actions of during a state recall before it starts executing any other components. The finish recall component is a component that the recall component forces HP ITG to execute 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 driver
  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. This will tell HP ITG 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 driver, the finish recall component should be the last

component in the driver.

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: A component must be set before another component to avoid side effects, but if it is placed before the other component in the driver, 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 flipped.

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

Solution: Use the IF RECALLING or IF PANELMODE, etc. constructs 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. Then, when HP ITG gets to the component, it will automatically execute its set actions.

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

Solution: In the finish recall component, do a set of the component. A set is used here because in the case of a recall, HP ITG is the controller and the instrument is being controlled.

FINISH

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.

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 that contains information about the current state of the instrument in it. For drivers that use learn strings, there will be no problems with state recalling because HP ITG can simply feed the learn string back to the instrument rather than execute every set action list of every component that changes in the new state. This eliminates any order of execution problems that could occur if learn strings were not used. To decide whether or not to use learn strings in a driver, consider the following concerns:

learn strings... PROs

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 driver 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 driver.

learn strings... CONs

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

If learn strings are a good decision for a particular instrument, here is the procedure to add them to a driver:

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

1. You will need a way to inform HP ITG 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 HP ITG needs to know the names. Add the following lines and components to the driver:

```
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. HP ITG will automatically execute the store component's set actions 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. See the section on developing the set/get actions for discrete components in "WRITING A DRIVER" for more information.

4. HP ITG executes the set actions of the recall component every time there is a state recall. Therefore, there needs to be a set actions list in the recall component that tell HP ITG 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 HP ITG from executing any other set actions in the driver, as usually happens in a state recall. When HP ITG starts recalling a state, it marks the components that change their value as invalid, then, before it starts executing set actions of invalid components, it executes the set actions of the recall component. The validate all statement will inform HP ITG that after the learn string has been sent to the instrument, all the components in the driver will be valid.

Manual Testing

Manual testing can be an important part of testing the driver. The test programs will detect many errors, but they cannot be designed to cover all the possible states without running for long periods of time. With a little knowledge of the instrument's operation and what the tricky parts of the driver are, it is possible to verify the driver's operation in relatively little time.

One method of manual testing is relatively straightforward. By just playing with the driver many errors will pop right out and can be corrected faster than by running a lengthy test. This method can be used with state recalls, also, to test the driver's ability to accurately recall one state from another.

With manual testing, errors need to be searched out. That is, you won't know anything is wrong with a component in another panel until you switch to that panel. Once an error is detected, there is also some question as to when the component value stopped matching the instrument control. To speed up the error detection and to guarantee that all inconsistencies will be detected at a given time, the tester_compare subroutine that is generated with test_gen can be added to the driver (temporarily) by using it as a usersub. Then, when you change a component value, you can click on a tester_compare button that you add (usually on a static front panel) and the tester_compare subroutine will go to work to compare the actual instrument controls to the expected values in the components of the driver. This allows you to change any component(s) in the driver in any order and then quickly find and detect any errors that may have occured.

The procedure for adding tester_compare to a driver is as follows:
(for RMB/UX only)

1. Generate the test program like you would for panelmode testing. You will end up with an ASCII program listing consisting of a main program, and three subroutines: tester_compare, read_instrument, and tester_error.

2. Use vi or another HP-UX editor to make the following deletions:

   Remove all lines with: COM /Tester/ (6 in all)
   Delete all calls to Hpt_monitor (3 in all)

3. Copy the line: Hpt_assign .....
   and all the lines: Hpt_assigncomp .....

   from the main program into the tester_compare subroutine right after the 'Tester_compare: !' label.

4. Ensure that the device name in the Hpt_assign statement, above, matches the instrument name in the workfile.

5. In subroutine tester_compare, declare a new variable:
   INTEGER Test_device

because Test_device is a variable used in the main program lines that are now added to the tester_compare subroutine.

6. Add the lines:
   SUB Hpxxxx_init
   SUBEND

   to the beginning of the file, where xxxx is the device name.

7. Save the edited file, exit vi and use the following awk scripts to unnumber and re-number the edited file. (BASIC does not like line numbers in the wrong order:

unnumber:
```
awk '{if($1~/^[0-9]+/) $1=""; print $0}'
```

number:
```
awk 'BEGIN{lineno=1;}{if($1!~/^[0-9]+/)  print lineno++ "  " $0}'
```

8. Enter the basic editor, and load the ASCII program and store the file in binary format using the STORE command, calling it USHPxxx where xxx is the same name as your driver.

9. Load HPITG

10. In the driver, put the command USERSUB TESTER_COMPARE in the set actions of the component you want to call tester_compare from.

11. Run HPITG and re-load the modified driver, remembering to use the same name as you used in steps 4, 6 and 8.

If all goes well, you can now test the driver manually and use the tester_compare subroutine to verify that the current state is valid. This method of manual testing allows you to check the results of your actions much quicker than visually comparing the components from panel to panel.

Usersubs in a driver are a bad idea. Make sure that the tester_compare usersub created above and the component that called it are removed from the driver after manual testing has been completed.

Interpreting Test Results

Driver testing is finished when you are confident that it meets the following specifications:

- The runtime operation of the driver is consistent with the operation of the instrument.

- Storing and recalling states operate correctly in the driver.

- The driver generates efficient, correct and fast code. Fast code, however, is somewhat a function of the driver action code itself.

- The panelmode operation of the driver is consistent with the operation of the instrument.

Note that the operation of the driver should be consistent with the operation of the instrument at all times. This does not mean there can be no errors in the test results, however. Not all errors uncovered during testing indicate problems with the driver. In some cases the errors uncovered during testing are actually from the instrument. (See examples below) Sometimes, errors that come from the instrument that do not result in tester_compare errors in the test program indicate that the driver is pushing the instrument to its limits and things are still working correctly. However, THIS IS ONLY TRUE IF THERE ARE GET ACTIONS IN THE COMPONENT(S) IN QUESTION.

A sample of the results file from a test is as follows:

```
Trace: test OFFSET_CH1
Error:    execute: Device: HP54502A,707 reported error -212
  Trace: in Tester_compare
Test OFFSET_CH1 component TIME_SENS: Instr was .0001, ITG expected .0001
Test OFFSET_CH1 component TIME_WINDOW_SENS: Instr was .0001, ITG expected .0001
  Trace: in Tester_compare
Error:    execute: Device: HP54502A,707 reported error -212
  Trace: in Tester_compare
```

This file shows that the instrument passed the test. There are four errors in the results. Two of the errors come from the instrument and two come from the subroutine tester_compare in the driver. The tester_compare errors are due to a difference in rounding between the instrument and HP ITG. The other two errors, error -212, are the result of the test program trying to set a control on the instrument out of range. Since the error is not accompanied by driver error from tester_compare, the state is still valid. This illustrates a good point: the driver does not need to be written to lock out invalid ranges or commands as long as the state will still be valid after the illegal operation. In this specific case, the reason the state is still valid is because the component OFFSET_CH1 uses a GET OFFSET_CH1 command in its panel set actions. This acts as a verification of the component value, and therefore the driver does not need to worry about locking out illegal values.

The testing phase is where most couplings are detected. Couplings are a situation in which the value of one control/component may change when the value of another component is changed. During testing, the important components in the driver are set to their minimum, intermediate and maximum values and then the values of all the components in the driver are compared to their corresponding controls on the instrument.

As an example of a coupling problem, the following driver was written:

```
REVISION 1.1;

COMPONENT A;
  TYPE INTEGER;
  VALUES RANGE 0,100;
  INITIAL 0;
END COMPONENT;
```

```
COMPONENT B;
  TYPE INTEGER;
  VALUES RANGE 0,100;
  INITIAL 0;
  SET ACTIONS;
    FETCH B;
    STORE A;
  END ACTIONS;
  GET ACTIONS;
    FETCH A;
    STORE B;
  END ACTIONS;
END COMPONENT;

PANEL ROOT;
  SIZE 214,97;

  TEXT 35,35 "A";
  CONTINUOUS A;
    POSITION 10,10;
    SIZE 60,19;
  END CONTINUOUS;

  TEXT 105,35 "B";
  CONTINUOUS B;
    POSITION 80,10;
    SIZE 60,19;
  END CONTINUOUS;
END PANEL;
```

Component A can be set to anything, but component B's get actions put the value of A into B. Therefore, component B's value should change every time component A's value does for the "state" to be valid. (This example does not use an instrument, so it is not an actual coupling problem, but if an instrument did exist with the two controls A and B, the get actions of B would query the instrument and return with the same value) Since B's value is dependent on A, there should be a COUPLED B statement in component A.

The following test_help output was generated for the driver, and the test results, after running test_gen, are also shown:

```
OPTIONS USEGETS TRACE
TEST A
  SET A 0
  COMPARE
  SET A 50
  COMPARE
  SET A 100
  COMPARE
ENDTEST
!
TEST B
  SET B 0
```

```
  COMPARE
  SET B 50
  COMPARE
  SET B 100
  COMPARE
ENDTEST
!

Trace: test A
  Trace: in Tester_compare
Test A component B: Instr was 100, ITG expected 0

  Trace: in Tester_compare
Test A component B: Instr was 0, ITG expected 50

  Trace: in Tester_compare
Test A component B: Instr was 50, ITG expected 100

Trace: test B
  Trace: in Tester_compare
  Trace: in Tester_compare
  Trace: in Tester_compare
```

The test results are typical for a coupling error. Notice that every time component A is changed, tester_compare returns an error that component B's value is invalid. The solution is to change component A to the following:

```
COMPONENT A;
  TYPE INTEGER;
  VALUES RANGE 0,100;
  COUPLED B;
  INITIAL 0;
END COMPONENT;
```

The coupled statement will notify HP ITG that every time component A is changed the get actions of component B must be executed in order for the state to be valid. In much the same way, any time you get a similar situation in a test results file the solution is most likely a coupling problem.

Note that in runtime mode operation, HP ITG does not execute couplings. If the test results were obtained in runtime operation, the proper solution would be a GET B statement somewhere in component A's set actions. The proper place to put this statement can depend on the nature of the set actions, although it is usually ok to put it at the end.

VI. Writing a good helpfile

A good helpfile should be concise but informative. Follow these guidelines when writing the helpfile:

- Include an overview of the instrument the driver was written for, along with a listing of the key instrument features.

- Include an overview of the main panel of the driver, any controls on the main panel, and any special or different features of the driver. If the driver uses a learn string, mention it here. Include a listing of the subpanels for the driver.

- Include a separate helpfile topic for every subpanel. Each control on the subpanel should be briefly described, along with a description of the subpanel operation at the beginning of the help entry.

- Include a list of error numbers and the corresponding error messages that the instrument may send.

- Include a list of all the components that correspond to controls on the instrument. This excludes dummy components, the charstring component, or others that a customer will never interact with.

- If there is a particular error that is extremely common in the driver, include a help topic on it. See the HP 3437 helpfile for information on "Triggering Too Fast" for an example.

Each help file entry is separated with a HELP .. ENDHELP construct. An example of the first few helpfile entries from the HP 54111D driver is:

```
HELP Overview
The HP 54111D is a high-performance
digitizing oscilloscope with the following
key features:

* 1 gigasample/second digitizing rate
* 2 channel simultaneous acquisition
* 500 MHz repetitive bandwidth
* 250 MHz single-shot bandwidth
* 8-bits vertical resolution
* 8k-deep waveform memory
* Autoscale for automatic setup
* Automatic waveform measurements
* General-purpose input coupling
* Waveform math functions
* Full color display
END HELP

HELP Using the Panel
This driver for the HP54111D provides a panel
that is divided into two parts:

1. An XY display appears on the panel's left
   side.  The following instructions explain
   how to use the display.

   - Click on the channel or channels you
     want HP ITG to display.  When the box
     which contains the channel number (e.g.,
     CH1) is highlighted, the channel is on
```

and HP ITG displays the waveform input to the channel.

- Click on the XY display itself to upload all displayed channels to HP ITG.

- Clicking on AUTOSCALE directs HP ITG to quickly set up the instrument to view a waveform.

- Turning Refresh off instructs HP ITG to wait until you click on the XY display before it displays a new waveform. With Refresh off, HP ITG keeps the old waveforms on the XY display until the waveform data changes, and then HP ITG will INVALIDATE all waveforms. Vmarkers and Tmarkers will stay on the XY display but their positions will not necessarily match that of the oscilloscope.

- Turning Refresh on tells HP ITG to automatically update the displayed channels after they have been modified. This means any old data will be written over. When you are working with single shot data or other important data, first turn Refresh OFF!

- A convenient REPETITIVE/REALTIME display is included to allow you to find out quickly what mode of operation the oscilloscope is in.

2. The following subpanels incorporate most of the features of the HP 54111D. The subpanel setup is very close to the front panel operation of the HP 54111D.

- Channel 1 Panel
- Channel 2 Panel
- Timebase Panel
- Trigger Panel
- Display Panel
- Delta V Panel
- Delta T Panel
- Memory Panel
- Function Panel
- Measure Panel
- Utility Panel
- Status Panel
- About Panel

NOTES

Bus communication between the oscilloscope and this driver is done using "HEADER OFF" and "LONGFORM OFF". Changing either of these during programmatic use could result in bus problems.

The menus on the oscilloscope's display will change as different subpanels are selected on the panel. In general, the subpanels are organized in the same manner as the oscilloscope's menus.

CAUTION

While this driver does not use a learn string for storing and recalling states, care should still be used when creating states without the oscilloscope present to ensure a valid state.

END HELP

HELP Channel Panels
These subpanels allow you to set up the following HP54111D features for either Channel 1 or Channel 2:

* Volts/Division

* Vertical Offset (volts)

* Presets (TTL,ECL,None) allows you to quickly set up the HP54111D to measure TTL or ECL logic levels.

* Coupling (DC,AC,Ground,DCFifty) allows your choice of input coupling.

END HELP

APPENDICES

APPENDIX A - "AXIOMS" IN DEVELOPING A DRIVER

1. States (valid ones) can only be guaranteed with a live instrument.

2. Programmatic model to keep in mind in RUNTIME, not panels mode.

3. Remember, a driver is meant to passively reflect the state of the instrument. The driver should be written to allow the instrument to respond naturally to inputs, not to lock out inputs that are known to illegal. If the instrument receives an illegal input, it will issue an error message and the driver should reflect the message to the user.

4. USERSUBs are to be avoided at all costs!

5. Once an error has occured in the driver, the driver may not reflect the current state of the instrument. *FINISH* What to do???

FINISH
....anything else?

APPENDIX B - OPERATING ENVIRONMENTS - Unix vs. Workstation

FINISH

APPENDIX C - PROGRAMMING PRACTICES

The Stack

The HP ITG stack operates much like an HP calculator. There is a stack, and a choice of operations to perform on the values in the stack. For those not familiar with the stack operations, see the System Documentation.

There is another stack HP ITG uses when it encounters GOSUB statements in a driver. This stack is separate from the math stack and is 10 levels deep. Therefore, the stack can handle recursion of up to 10 levels nesting. The following example, taken from the HP 54501 Digitizing Oscilloscope driver, shows the use of recursion in a driver:

```
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;

ACTIONS FLUSH_ERR_QUEUE;

  ! Routine to flush the error queue in the 54501 -- note that the queue can
  ! be up to 20 elements deep, but because the ITG recursion stack is limited
  ! to 10 deep a double 'pop' per pass is required ...

  OUTPUT STRING "SYSTEM:ERR?;";
  ENTER STACK FORMAT K; ! reads error number
  SELECT STACK;
  CASE 0;
    ! no further errors -- queue is empty ...
```

```
  CASE ELSE;
   OUTPUT STRING "SYSTEM:ERR?;";
   ENTER STACK FORMAT K; ! reads error number
   SELECT STACK;
   CASE 0;
    ! no further errors -- queue is empty ...
   CASE ELSE;
    GOSUB FLUSH_ERR_QUEUE;
   END SELECT;
  END SELECT;
END ACTIONS;
```

The error number is entered into the dummy component ERR_NUMBER, and the error queue stack in the 54501 is emptied by the routine FLUSH_ERR_QUEUE by the use of recursion. Since the instrument error queue can be up to 20 elements deep, the routine must pop two values per pass because ITG's recursion stack is only 10 elements deep.

Use of panel set actions for verification

In some more complicated drivers the range values of components change almost as often as the component values themselves, requiring components to be able to predict what the legal range is whenever they change. This requires a lot of programming time and space, space that complicated drivers cannot spare much of. A common practice used in drivers that use learn strings, is to execute a GET of the component in the panel set actions. This is just a verification of the component value to ensure that the driver's state matches the instrument's state.

Taken from the HP54111D driver:

```
COMPONENT SENSITIVITY_CH1;
 TYPE CONTINUOUS;
 VALUES RANGE 1E-6,20E3;
 INITIAL 2;
 COUPLED OFFSET_CH1,MARKER_MIN_MAX,TRIGGER_LEVEL_CH1;
 SET ACTIONS;
  OUTPUT STRING "CH 1 SENS ";
  OUTPUT SENSITIVITY_CH1 FORMAT 'K,";"';
  FETCH RECALLING; NOT;
  IF STACK THEN;
   FETCH (PRESET_CH1)NONE;
   STORE PRESET_CH1;
  END IF;
 END ACTIONS;
 GET ACTIONS;
  OUTPUT STRING "CH 1 SENS?";
  ENTER SENSITIVITY_CH1 FORMAT K;
 END ACTIONS;
 PANEL SET ACTIONS;
  GET SENSITIVITY_CH1; ! <---------- Verification
  GOSUB UPDATE_TRACE_DATA;
 END ACTIONS;
```

END COMPONENT;

Common action lists between components

Sometimes several different components will have the same set or get actions. To save memory and to make the driver shorter, the use a common action list is a good idea. The action list is separate from any component in the driver, and it can be placed anywhere in the driver. To call the action list from another action list, the gosub statement is used. Note: to save more memory, the following two constructs are equivalent:

```
COMPONENT FOO;                    COMPONENT FOO;
  TYPE INTEGER;                   TYPE INTEGER;
  INITIAL DONTCARE;                 INITIAL DONTCARE;
  SET ACTIONS;            ----->  SET ACTIONS ACTION_LIST;
    GOSUB ACTION_LIST;              END COMPONENT;
  END ACTIONS;
END COMPOENNT;
```

Not only does the shorter component save driver code, but the amount of memory used when the component is parsed is also less.

APPENDIX D - Tools: hpidc, awk, make, m4, cpp, etc.

FINISH

APPENDIX E - test_help and test_gen manual page entries

TEST_HELP(1L) TEST_HELP(1L)

NAME
test_help - produce an initial ID test file

SYNOPSIS
test_help ID_file [output_file]

HP-UX COMPATIBILITY
Level: HP-UX/Local

Origin: MSO

DESCRIPTION
test_help reads an ID file and produces a first shot at the test language to test this ID.

It is expected that the user will then edit the output file, filling in test cases which make sense, and removing those which don't. Warnings are generated in some places where it's obvious to test_help that it has no idea what to do.

NOTE
There are 3 known problems with test_help. First, it doesn't understand quoted strings in values range lists. It will

leave the quotes in place and seperate parameters at white space. You will have to fix these by hand. (Hint: search for "" in vi.) Also, test_gen will not handle quoted strings either, so if you need a quoted string, you'll have to modify the generated RMB. Please see me if this causes too much trouble, because it could be fixed with some work.

Second, nawk gets floating exceptions when trying to compute with numbers greater then 1E9. This happens if you have a continuous component with limits larger than this. You can remove the offending line(s), generate the test code, and fix the ranges by hand, or just change the range to something smaller during test_help. This would be difficult to fix.

Thirdly, some components have other components as their range limits. At best, it will pass these names through unchanged. At worst, this will crash it. I suggest trying this unchanged, and then going back and fixing the test points by hand. Interestingly, the ID's I've tried have generated legal test code, and since test_gen provides an RMB variable name with the same name as the component, the range limit might really be what you want to test. However, look it over.

BUGS

in addition to the above, probably lots.

SEE ALSO

test_gen(1L)

NAME

test_gen - produce RMB test code for an ID

SYNOPSIS

test_gen test_file ID_file device [RMB_output]

HP-UX COMPATIBILITY

Level: HP-UX/Local

Origin: MSO

DESCRIPTION

test_gen reads an ID file and the test description file and

produces RMB test code to exercise RAZOR and this ID. Device is the name given to the instrument when the ID was read into RAZOR.

The user will probably want to use test_help to generate the initial test description file, edit this file by hand, and then feed it to test_gen.

GENERAL PROVISIONS

test_gen will set up a main RMB program, provide an RMB variable of the proper type with the same name as every component, and keep the internal ID of each component in an INTEGER named H_<component name>.

It also generates a subprogram named Tester_compare whose job is to peek the current state vector, call Read_instrument, and check to see if the state vector has changed.

If OPTIONS USEGETS is specified, Read_instrument is also generated which does a Hpt_get on every component. If you trust your GET ACTIONS, this is the way to go.

OPTIONS RECALL adds a whole section of recall tests to your test program. Every time you perform a COMPARE, the state of the instrument will be saved. At the end of your test suite, all the saved states will be recalled and an automatic COMPARE performed. The idea here is to test recall state for your ID.

The automatic recalls are performed in a pattern which exercises every possible from and to state. This is not necessarily exhaustive from an ID testing standpoint, but it isn't bad for a little bit of work, and it does make the test run for hours (or days).

One class of ID bugs is where a component value is clobbered by another during the recall. OPTIONS RECHECK adds a test to detect this condition. When specified, a compare is made

between the current RAZOR state and the stored state immediately after the call to Hpt_recall. OPTIONS RECHECK implies RECALL. Note that the Tester_recheck sub must be loaded by hand. See RECHECK below.

Read_instrument can be modified by the user to enter binary state information from the instrument and poke it into RAZOR. This is the recommended testing approach, both for speed and best test coverage. You can test your GET ACTIONS by running the test program both ways.

A subprogram named Tester_error is generated to report errors. The user can modify this to suit his needs. If you do, you'll have to delete the generated sub each time, and load in your own version.

TEST CODE SYNTAX

<program> ::= [<options>] [<settings>] [<ignore>] [<test>...]

<options> ::= OPTIONS [TRACE] [USEGETS] [RECALL] [RECHECK]

TRACE adds print statements at several interesting places in the generated code, and turns Hpt_monitor on for this device.

USEGETS generates the Read_instrument subprogram which uses the ID's GET ACTIONS to read the instrument state. This is also a good way to get a copy of Read_instrument which you can modify.

RECALL generates code to do the recall testing. This will run for quite a while.

RECHECK generates code to see if the razor state matches the RECALL'ed state during the recall testing. Specifying this implies RECALL.

If you specify OPTIONS RECHECK, you must do a LOADSUB ALL FROM "RECHECK" in order to load the Tester_recheck subprogram. A copy of this is available in the razor_tools directory on hpislx.

<settings> ::= [<workfile>] [<printer>]

<workfile> ::= WORKFILE <filename>

where <filename> is the name of the workfile. WFDFLT is the default. Do not put the filename in quotes.

<printer> ::= PRINTER <print_spec>

where <print_spec> is the what will follow a "PRINTER IS" command in the generated code. This allows you to redirect all test output. If this is going to a file, the filename must be in quotes.

<ignore> ::= IGNORE <component>

IGNORE causes test_gen to ignore the named component during COMPARE operations.

<component> is any component in your ID. Long component names may be truncated.

<test> ::= TEST <test_name>
[<test_stmt>...]
ENDTEST

<test_name> is the test name.

<test_stmt> ::= <recall> |
<set> |
<get> |
<peek> |
<poke> |
<compare> |
<expect> |
<basic>

<recall> ::= RECALL <state>

<state> is a stored RAZOR state for this instrument.

<set> ::= SET <component> <value>

<value> is a value suitable for the named component. On components expecting strings, do not put quotes around the value. It is not possible to specify values for array components.

<get> ::= GET <component>

The value is left in the RMB variable of the same name as <component>. Iparm contains the VALID(1) / INVALID(-1) / DONTCARE(2) status.

<peek> ::= PEEK <component>

See <get>.

<poke> ::= POKE <component> <value>

See <set>.

<compare> ::= COMPARE

COMPARE calls Tester_compare to see if the instrument state matches the state stored in RAZOR. This assumes that the

subprogram Read_instrument exists and sets the complete RAZOR state to the current instrument settings. If something cannot be read from the instrument, this test will not detect errors in RAZOR's idea of that component.

<expect> ::= EXPECT <component> <e_value>

<e_value> ::= <value> | VALID | INVALID | DONTCARE

EXPECT tests to see if the named component has the expected value. It is useful in hand testing expected component interactions. See <set>.

<basic> is anything test_gen doesn't understand. It assumes that it's inline RMB, and passes it unchanged into the generated RMB program.

NOTE

There are 2 known problems with test_gen. First, it doesn't understand quoted strings in component values. It will leave the quotes in place and throw away anything after white space. If you really need quoted strings, you'll have to patch the RMB, or just put inline RMB in the test description file in the first place. Also, use this technique any time you want to do something not supported by test_gen. This could be fixed, but it will take some work.

Second, the component names in the test description file must be upper case only. test_help will upshift all component names when the test file is first generated, but the user will have to maintain this when modifying the file. Again, this can be fixed with some work.

BUGS

in addition to the above, probably not quite as many as test_help.

SEE ALSO

test_help(1L)

APPENDIX F - Blank design sheet

Control:
HP-IB command:
command? query?
Valid menu:
Valid mode:
Type:
Range of values:
Initial value:
Survive a reset?

Control:
HP-IB command:
    command?   query?
Valid menu:
Valid mode:
Type:
Range of values:
Initial value:
Survive a reset?

Control:
HP-IB command:
    command?   query?
Valid menu:
Valid mode:
Type:
Range of values:
Initial value:
Survive a reset?

APPENDIX G - Communicating between drivers

There are two new features for communicating between drivers and
for allowing a driver to communicate with more than one instrument.

An ID writer can now reference a component in another ID for the following
keywords:

FETCH
STORE
SET
GET
IF
SELECT
ENTER
OUTPUT

The syntax for referencing a remote component is

<component-name> @ <id-name>

where id-name is the name of a string component which holds the logical
name of the remote ID. For example, if want to change the frequency of
the HP3314A, which the user called "FUNC_GEN" in the configuration panel,
from another ID, we could write

STORE FREQ@FG;

where we would have component FG defined like this:

```
COMPONENT FG;
    TYPE STRING 25;
    INITIAL "FUNC_GEN";
END COMPONENT;
```

We could also say

```
IF FREQ@FG THEN
.
.
.
```

We could also say

```
FETCH (FREQ@FG) ACV;
```

in order to fetch the discrete value ACV for the remote component "FREQ".

This syntax of "<component-name> @ <id-name>" works for all of the above keywords with the following caveats:

SELECT COMP@ID will always assume that the component is numeric, and will give an error if it is a string

OUTPUT COMP@ID and ENTER COMP@ID will not work with array components.

To write a driver that communicates with more than one device, the ID writer could code

```
FETCH ADDR;
STORE OLD_ADDRESS;

FETCH 705;
STORE ADDR;
```

which would save the old address in the continuous component old_address, and change the address to 705. The "STORE ADDR" does an implied FLUSH.