Measurement Guide and Programming Examples

Agilent Technologies PSA Series Spectrum Analyzers

This guide documents firmware revision A.03.xx
This manual provides documentation for the following instruments:

E4440A (3 Hz – 26.5 GHz)
E4443A (3 Hz – 6.7 GHz)
E4445A (3 Hz – 13.2 GHz)
E4446A (3 Hz – 44.5 GHz)
E4448A (3 Hz – 51.0 GHz)

Agilent Technologies

Manufacturing Part Number: E4440-90138
Supersedes: E4440-90063

Printed in USA

October 2003

Where to Find the Latest Information

Documentation is updated periodically. For the latest information about Agilent PSA spectrum analyzers, including firmware upgrades and application information, see: http://www.agilent.com/find/psa.

Contents

Contents

1 The Basics

Assumption You know the basics of spectrum analyzer operation, and the location and function of front and rear panel keys and connectors. If *not*, refer to the Getting Started guide.

NOTE In this manual, preset means *factory* preset.

This chapter contains basic information about using the analyzer.

- "Using Files" on page 7

For detailed information on analyzer functions, refer to the Reference guide.

Using Files

This section provides information on how to work with files.

Assumption

The information in this section is provided with the assumption that you know how to save a file, and how to locate and view catalogs and files. If you do not, refer to the Getting Started Guide for details.

In this section, you will find the information on the following:

Creating a Directory (or sub-directory)

You can add a directory or sub-directory to either the `A:` floppy disk or the internal `C:` drive.

1. Open the `Create Directories` menu: press **File**, **More**, **Create Dir**.
2. Navigate through the file system until the `Path:` field displays the desired directory.
3. Press **Name** and use the Alpha Editor to enter the desired name for the new directory. To terminate the entry, press the **Enter** front panel key.
4. To create the directory, press **Create Dir Now**. Once the directory is created, the status bar displays:
   `Directory` <*path*><*name*> `created`

The Basics

Deleting Files

You can delete individual files from *any* directory, as described in the following procedure; you can also delete *all* files and directories from a floppy disk at one time (see page 9).

Deleting One File

1. If you are deleting a file from a floppy disk, ensure that the disk is *not* write protected, then place the disk in the analyzer's floppy drive.
2. Open the `Delete` menu: press **File**, **Delete**.
3. Select the type of file you want to delete: press **Type**, then select the type you want from the `Type` directory.
4. Select the drive and directory that contains the file you wish to delete (the currently selected location appears in the `Path:` field):

   Press **Dir Select**, highlight the desired directory and press **Dir Select** again. Continue until you have located the desired directory.

NOTE If you are not familiar with how to move among directories and locate files, refer to the Getting Started guide for details.

5. Highlight the file you want to delete.
6. Press **Delete Now**. The pop up message `Deleting file` appears on the display during the operation. When complete, the status bar displays the message: <*path*><*filename*> `file deleted`., and the file no longer appears in the directory.

Deleting All Files and Directories from a Floppy Disk

Use the following steps to delete all previously stored data from a *pre-formatted* floppy disk.

1. Ensure that the disk is *not* write protected, then place it in the analyzer's floppy drive (A:\).
2. Press **File**, **More**, **Delete All**. The directory information box is active (highlighted), and displays the floppy disk volume ([-A-]).

   The files on the disk are *not* displayed at this point. You must use **File**, **Catalog** to see the files.
3. Press **Delete All Now**. The following message appears in the display window:
   `WARNING: You are about to destroy ALL data on Volume A:. Press Delete all again to proceed or any other key to abort.`

   To abort the process, press any key *other than* **Delete All Now**.
4. To delete all files and directories, press **Delete All Now** a second time.

   The message `Delete All` appears in the display window.
5. After all files and directories are removed, the following message appears in the status line: `Volume A: delete complete`. (If the disk is write-protected, the files will not be deleted even though it looks like it does.)

Loading a File

1. Reset the analyzer: press **Preset**, **Factory Preset** (if present).
2. Open the **Load** menu: press **File**, **Load**.
3. From the `Load` menu, select the type of file you want to load.

NOTE Not all file types can be loaded back into the analyzer: Screen files and CSV (comma separated value) cannot be loaded. Screen and CSV files are designed for use with a PC.

4. Select the directory where your file is located.
5. Select (highlight) the file you want to load into the analyzer.
6. For a state file, skip this step.

   For a trace file, select the trace into which you wish to load the file. For example, **Destination**, **Trace 2**.
7. Press **Load Now** to load the specified file. The status bar reads: <*path*><*file*> `file loaded`.

Key Points when Loading Trace Files

- Because the state of the analyzer is saved along with the trace, when the trace is loaded, all of the settings and annotations are restored to the values displayed when the trace was originally stored.
- The trace is loaded in View mode so that it does not update; the data remains on screen for printing, analysis, and so on.

Renaming a File

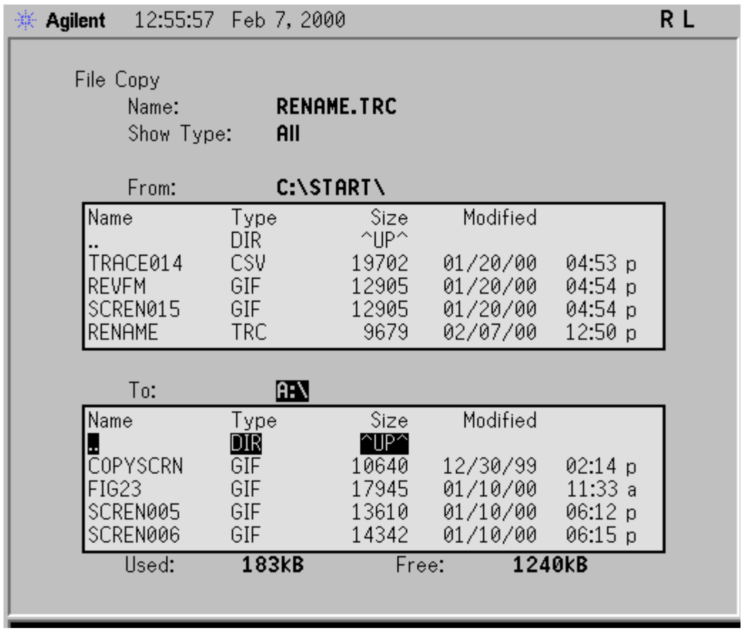
1. Open the `Rename` menu: press **File**, **Rename**.
2. Open the `Type` menu: press **Type**.
3. From the Type menu, select the type of file you want to rename.
4. Select the drive and directory where the file is located.
5. Select the file you want to rename.
6. Open the `Alpha Editor` menu: press **Name**.
7. Use the editor to rename the file (the `Name:` field is limited to eight characters), and press the **Enter** front panel key to terminate the entry.
8. Press **Rename Now**: the file is renamed and visible within the directory displayed on the analyzer. The status line displays the message:
   <*path*><*old filename*> `file renamed to` <*path*><*new filename*>

Copying a File

1. Open the `Copy` menu: press **File**, **Copy**.

   This menu displays two directory boxes, labeled `From:` and `To:` directly above the boxes. See Figure 1-1.

Figure 1-1 Copy Menu



2. Place a formatted 1.44 MB floppy disk into the A: drive.

NOTE Ensure that the disk is not write-protected.

3. Open the `Trace` menu: press **Type**, **Trace**.
4. From the `Trace` menu, select the type of file you want to copy.
5. If the **Dir** softkey does not have `From` underlined, press to underline it. This highlights the `From:` field (the directory from which you will copy)
6. Select the desired directory and highlight the file that you wish to copy.
7. Press **Dir** to underline `To`.

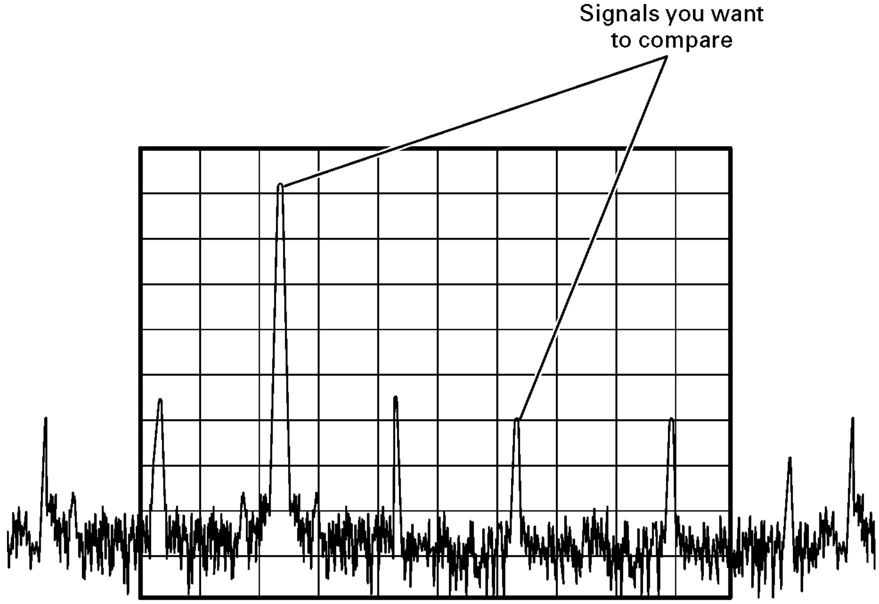
   The `To:` field highlights. This is the directory to which you will copy.
8. Select the desired directory and press **Copy Now**.

   The message `Copying file` appears. When complete, the status bar displays: <*directory*><*filename*> `file copied`.

2 Comparing Two Signals: Frequency and Amplitude

This chapter provides the following examples:

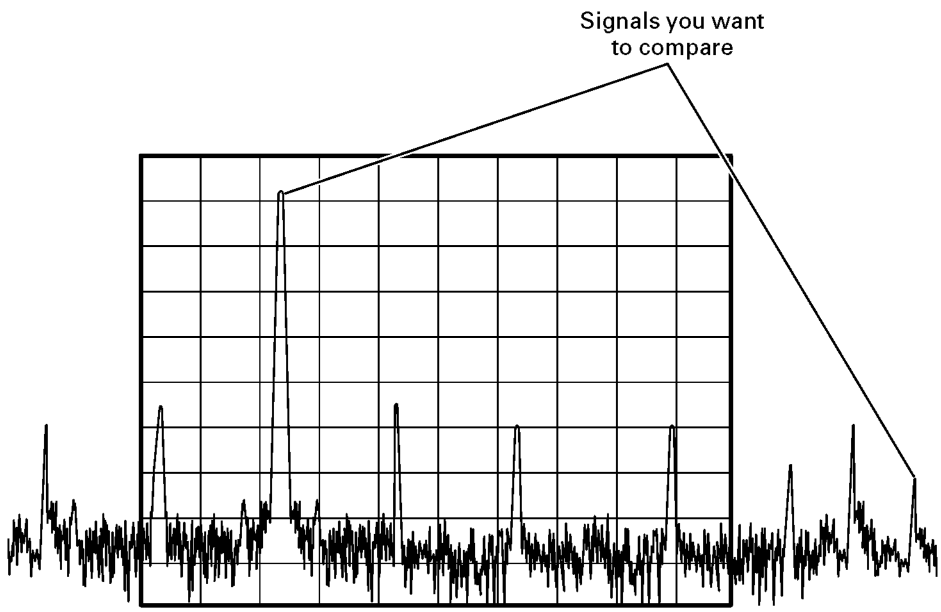
- “Comparing Signals on the Same Screen” on page 16



You can compare two signals whether they both appear on the screen at the same time (as shown above), or not (as shown in the following figure).

• "Comparing Signals not on the Same Screen" on page 20

The ability to compare signals when only one can be displayed at a time is useful for harmonic distortion tests, or any time narrow span and bandwidth are necessary to measure low-level signals.



Comparing Two Signals: Frequency and Amplitude

Comparing Signals on the Same Screen

Signals with Constant Levels (using Marker Delta)

1. Preset the analyzer, then set the following:

   - Reference Level: 10 dBm
   - Center Frequency: 30 MHz
   - Span: 50 MHz

2. Ensure that the rear panel 10 MHz output is on:

   Press **Input/Output**. Check the **10MHz Out** softkey. If **Off** is selected (underlined), press the key to select **On**.

3. Connect the analyzer's rear panel `10 MHz OUT (SWITCHED)` to the front-panel RF input.
4. Place a marker on the 10 MHz peak: Press **Peak Search**.
5. Anchor the first marker and activate a second marker at the same position: Press **Marker**, **Delta**.

   Note that the label on the first marker changes to `1R`, indicating that it is the reference point.

6. Use the knob to move the second marker (labeled `1`) to a different peak (for this example, the 20 MHz peak).

   Because delta marker is now the active function, both the active function block and the marker annotation display the amplitude and frequency *difference* between the markers, as shown in Figure 2-1.

7. Turn the markers off: Press **Marker**, **Off**.

NOTE

Alternate Methods

Replace the keystrokes in steps 4 through 6 with either:

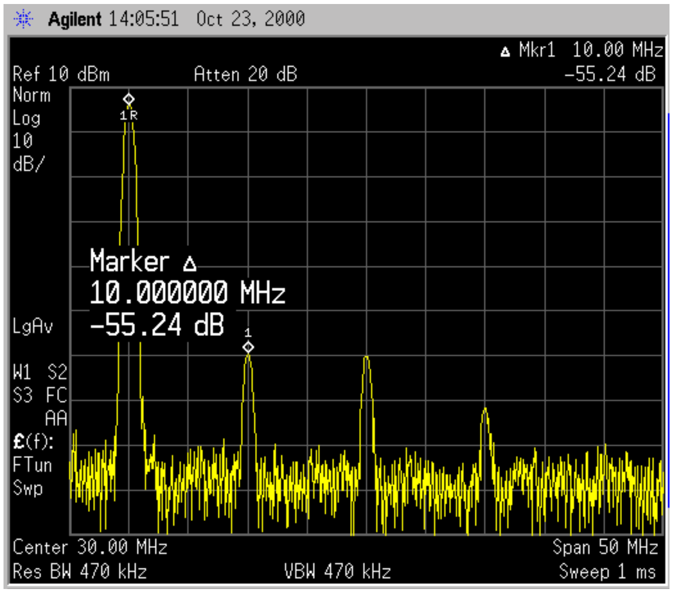
- Press **Sweep**, **Single**, **Peak Search**, **Marker**, **Delta**, **Return** (or **Peak Search**), **Next Peak**.

  (the **Return** hardkey is located directly below the softkeys)

  Or

- Press **Marker** and use the knob to position the marker. Then press **Marker**, **Delta** and position the second marker.

Figure 2-1 Reading the Marker Delta Value



Signals with Varying Levels (using Delta Pair)

The Delta Marker function (described on page 16) anchors the *reference* marker in both frequency *and* amplitude. The Delta Pair function, described in this example, enables the reference marker to remain on the trace, and lets you adjust either the reference marker or the delta marker, or both.

1. Preset the analyze, then set the following:

    - Reference Level: 10 dBm
    - Center Frequency: 30 MHz
    - Span: 50 MHz

2. With the rear panel 10 MHz output on (As described on page 16, in Step 2.), connect the analyzer's rear panel `10 MHz OUT (SWITCHED)` to the front-panel RF input.
3. Place a marker on the 10 MHz peak: Press **Peak Search**.
4. Anchor the first marker and activate a second marker at the same position: Press **Marker**, **Delta.**
5. Use the knob to move the second marker (labeled `1`) to a different peak (for this example, the 20 MHz peak).

    The marker annotation shows the difference between the two peaks.

6. Remove the signal from the input.

    Note that the reference marker remains anchored at the former frequency and amplitude of the 10 MHz signal. The delta marker stays on the trace and now shows the difference between the noise level at the delta frequency and the original amplitude of the 10 MHz signal.

7. Reconnect the signal, then reset the marker to a single marker on the 10 MHz peak:

    Press **Marker**, **Normal**, **Peak Search**.

    Activate a second marker at the same position *without* anchoring the first marker: Press **Marker**, **Delta Pair**.

8. Select the second marker: Press **Delta Pair** again, to underline Δ.
9. Use the knob to move the second marker (labeled `1`) to a different peak (for this example, the 30 MHz peak).

    Because delta marker is the active function, both the active function block and the marker annotation display the amplitude and frequency difference between the markers (just as when using the Delta Marker function, as shown in Figure 2-1).

10. Select the reference marker: Press **Delta Pair** to select (underline) **Ref**.

11. Use the knob to move the reference marker to the 20 MHz peak.

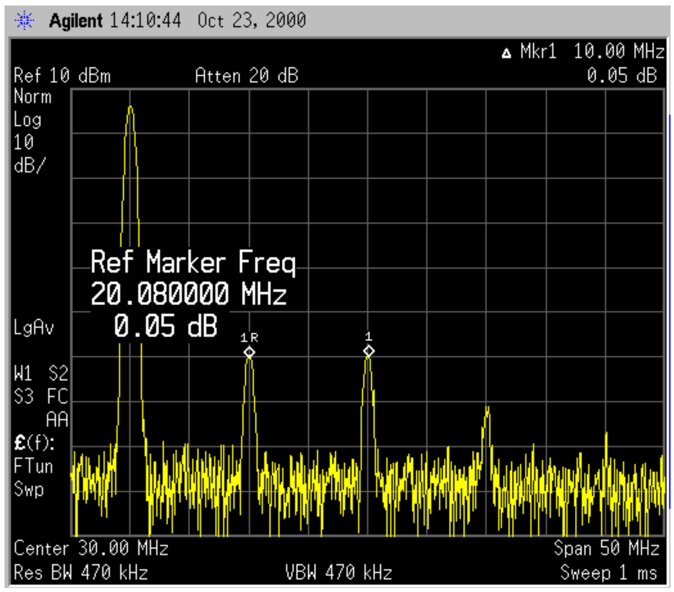
    Note that as you move the marker, it stays on the trace.

    Now the active function block and the marker annotation display the amplitude and frequency difference between the 20 MHz and 30 MHz peaks, as shown in Figure 2-2.

12. Disconnect the signal input. Note that *both* markers drop into the noise.

13. Turn the markers off: Press **Marker**, **Off**.

Figure 2-2 Reading the Marker Delta Value



Comparing Signals *not* on the Same Screen

1. Preset the analyzer, then set the following:

   - Reference Level: 10 dBm
   - Center Frequency: 10 MHz
   - Span: 5 MHz

2. With the rear panel 10 MHz output on (As described on page 16, in Step 2.), connect the analyzer's rear panel `10 MHz OUT (SWITCHED)` to the front-panel RF input.

3. Place a marker on the 10 MHz peak: Press **Peak Search**.

Setting Center Frequency Step Size

4. Set the center frequency step size equal to the marker frequency (in this example, 10 MHz): Press **Marker →**, **Mkr → CF Step**.

5. Activate the marker delta function: Press **Marker**, **Delta**.

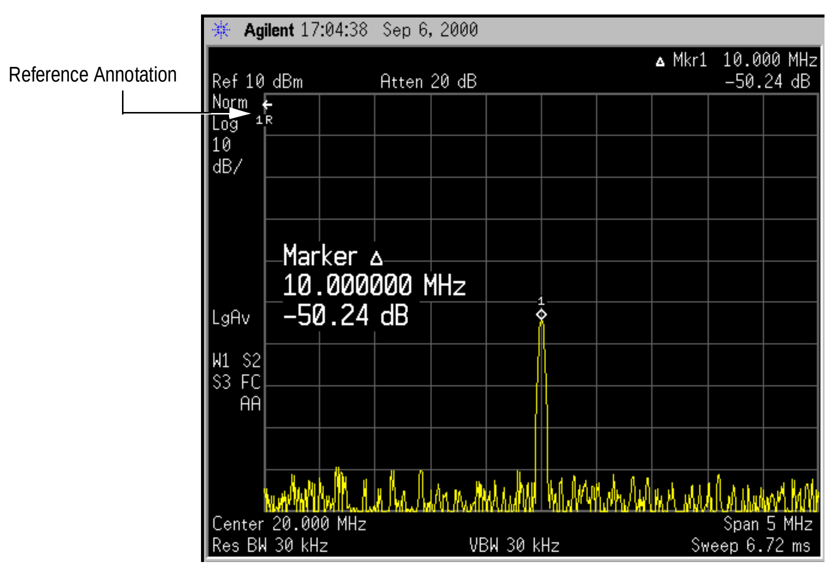
6. Increase the center frequency by 10 MHz:

   Press **FREQUENCY**, **Center Freq**, ⇑.

   Figure 2-3 shows the reference annotation for the delta marker (`1R`) at the left side of the display, indicating that the 10 MHz reference signal is at a lower frequency than the frequency range currently displayed.

   The delta marker appears on the peak of the 20 MHz component. The delta marker annotation displays the amplitude and frequency difference between the 10 and 20 MHz signal peaks.

Figure 2-3 Delta Marker with Reference Signal Off–Screen



3 Measuring a Low–Level Signal

The analyzer's ability to measure a low-level signal is limited by internally-generated noise. The measurement setup can be changed in several ways to improve the analyzer's sensitivity. Resolution bandwidth settings, when properly adjusted, affect the level of internal noise *without* affecting the signal amplitude.

This chapter provides the following examples:

- "Reducing Input Attenuation" on page 23

  The input attenuator affects the level of a signal passing through the instrument. If a signal is very close to the noise floor, reducing input attenuation can bring the signal out of the noise.

CAUTION Ensure that the total power of all input signals at the analyzer RF input does not exceed +30 dBm (1 watt).

- "Decreasing the Resolution Bandwidth" on page 25

  Resolution bandwidth settings affect the level of internal noise without affecting the signal level. Decreasing the RBW by a decade reduces the noise floor by 10 dB.

- "Using the Average Detector and Increased Sweep Time" on page 26

  When the analyzer's noise masks low-level signals, changing to the average detector and increasing the sweep time smooths the noise and improves the signal's visibility. Slower sweeps are required to average more noise variations.

- "Trace Averaging" on page 27

  Averaging is a digital process in which each trace point is averaged with the previous trace-point average. Selecting averaging changes the detection mode from normal (a type of peak detection) to sample, smoothing the displayed noise level. Sample mode displays the instantaneous value of the signal at the end of the time or frequency interval represented by each display point, rather than the value of the peak during the interval. Sample mode may not measure a signal's amplitude as accurately as normal mode, because it may not find the true peak.

Reducing Input Attenuation

CAUTION Ensure that the total power of all input signals at the analyzer RF input does not exceed +30 dBm (1 watt).

1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	300 MHz	• Reference Level:	–40 dBm
• Amplitude:	–80 dBm	• Center Frequency:	300 MHz
• RF Output:	On	• Span:	5 MHz

2. Connect the signal source to the analyzer's RF input.
3. Move the desired peak (in this example, 300 MHz) to the center of the display:

   Press **Peak Search**, **Marker →**, **Mkr → CF**.
4. Reduce the span to 1 MHz (as shown in Figure 3-1):

   Press **Span**, **1**, **MHz**.

   If necessary, re-center the peak.
5. Set the attenuation to 20 dB:

   Press **AMPLITUDE**, **Attenuation**, **2**, **0**, **dB**.

   Note that increasing the attenuation moves the noise floor closer to the signal level.

   A "#" mark appears next to the `Atten` annotation at the top of the display, indicating that the attenuation is no longer coupled to other analyzer settings.
6. To see the signal more clearly, set the attenuation to 0 dB (as shown in Figure 3-2).

CAUTION When you finish this example, increase the attenuation to protect the analyzer's RF input:

Either press **Attenuation** so that **Auto** is selected, or press **Auto Couple**.

Figure 3-1 Low-Level Signal

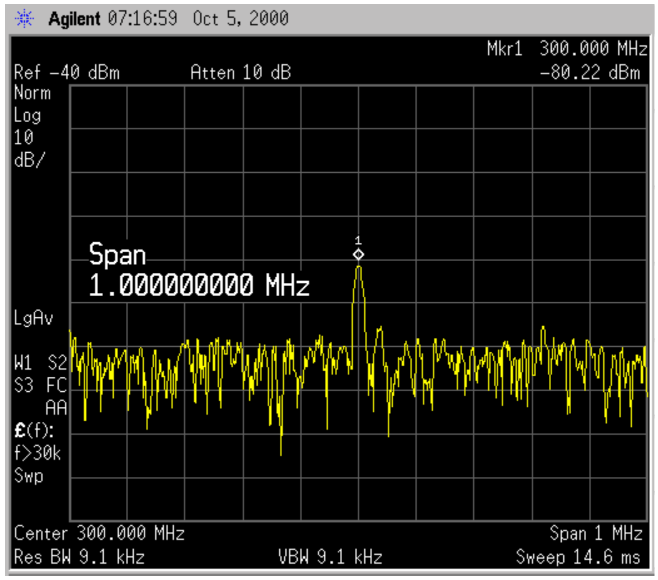
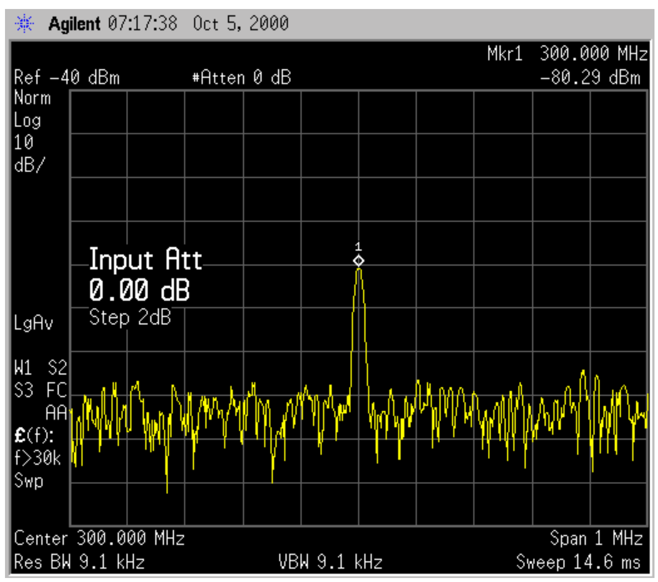


Figure 3-2 Using 0 dB Attenuation



Decreasing the Resolution Bandwidth

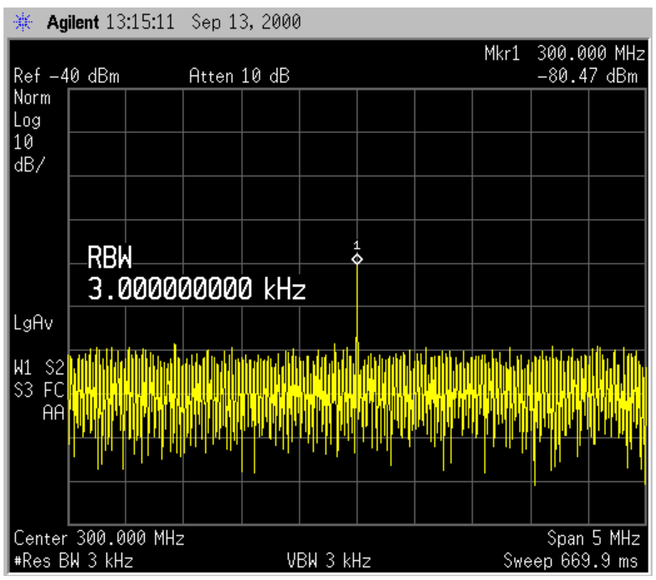
1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	300 MHz	• Reference Level:	–40 dBm
• Amplitude:	–80 dBm	• Center Frequency:	300 MHz
• RF Output:	On	• Span:	5 MHz

2. Connect the signal source to the analyzer RF input.
3. Decrease the resolution bandwidth: Press **BW/Avg**, ⇓.

   The low-level signal appears more clearly because the noise level is reduced (see Figure 3-3).

Figure 3-3 Decreasing Resolution Bandwidth



A “#” mark appears next to the `Res BW` annotation in the lower left corner of the screen, indicating that the resolution bandwidth is uncoupled.

RBW Selections

Using the step keys, you can change the RBW in a 1–3–10 sequence. Choosing the next lower RBW for better sensitivity increases the sweep time by about 10:1 for swept measurements, and about 3:1 for FFT measurements (within the limits of RBW).

Using the knob or keypad, you can select RBWs from 1 Hz to 3 MHz in approximately 10% increments, plus 4, 5, 6 and 8 MHz. This enables you to make the trade off between sweep time and sensitivity with finer resolution.

Using the Average Detector and Increased Sweep Time

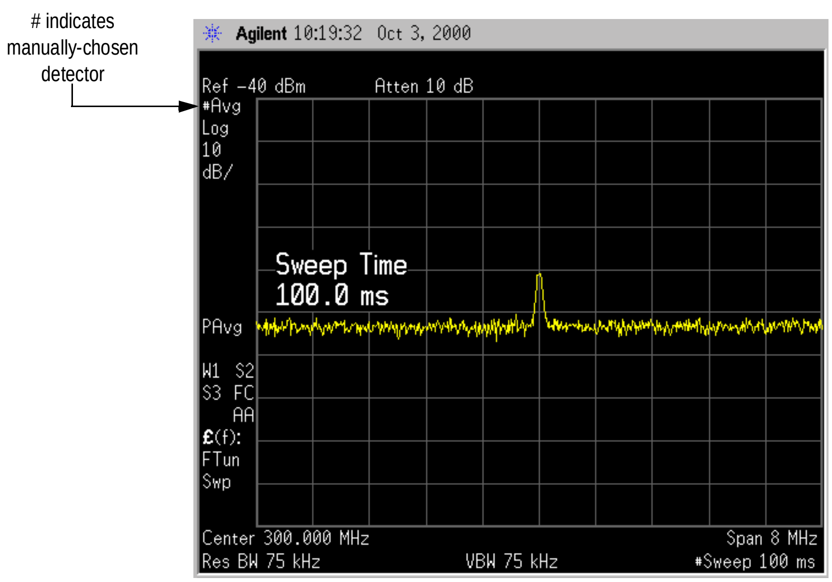
1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	300 MHz	• Reference Level:	–40 dBm
• Amplitude:	–80 dBm	• Center Frequency:	300 MHz
• RF Output:	On	• Span:	5 MHz

2. Connect the signal source to the analyzer's RF input.
3. Select the average detector: Press **Det/Demod**, **Detector**, **Average**.

   A "#" mark appears next to the Avg annotation, indicating that the detector has been chosen manually (see Figure 3-4).
4. Increase the sweep time and note how the noise smooths out, as there is time to average more noise values for each of the displayed data points: Press **Sweep**, **Sweep Time**, ⇑.
5. With the sweep time at 100 ms, change the Avg/VBW type to log averaging: Press **BW/Avg**, **Avg/VBW Type**, **Log-Pwr**.

Figure 3-4 The Effect of Sweep Time



Trace Averaging

Trace averaging is a digital process that averages each trace point with the previous trace-point average.

NOTE This is a trace processing function and is not the same as using the Average detector (as described on page 26).

1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	300 MHz	• Reference Level:	–40 dBm
• Amplitude:	–80 dBm	• Center Frequency:	300 MHz
• RF Output:	On	• Span:	5 MHz

2. Connect the signal source to the analyzer RF input.
3. Initiate video averaging: Press **BW/Avg**, **Average** (to select **On**).

   As the averaging routine smooths the trace, low level signals become more visible. `Average 100` (the default number of samples, or sweeps, to complete the averaging routine) appears in the active function block.
4. With average as the active function, set the number of samples to 25:

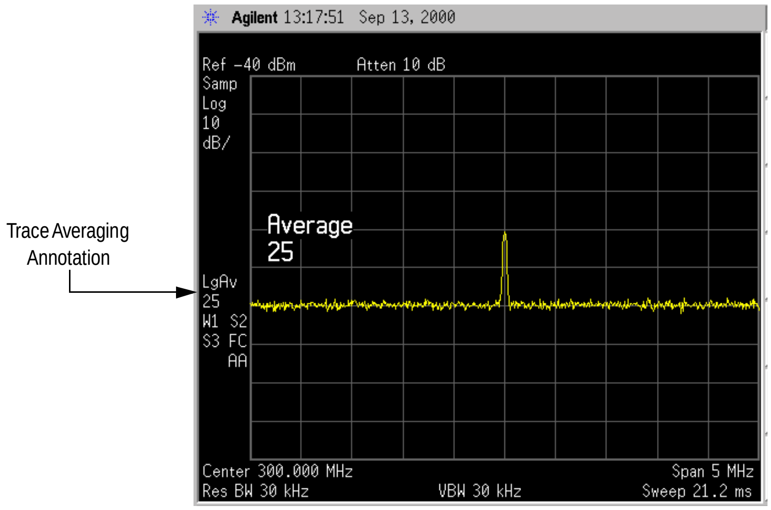
   Press **2**, **5**, **Enter**.

   Annotation on the left side of the graticule shows the type of averaging (LgAV in this example, as shown in Figure 3-5), and the number of traces averaged.

   Changing most active functions restarts the averaging, as does toggling the **Average** key. Once the set number of sweeps completes, the analyzer continues to provide a running average based on this set number.

NOTE If you want the measurement to stop after the set number of sweeps, use single sweep: Press **Sweep**, **Sweep** (to select **Single**), and then toggle the **Average** key.

Figure 3-5 Using Trace Averaging, Continuous Sweep



4 Resolving Signals

This chapter provides the following examples:

- "Separating Equal-Amplitude Signals" on page 32

  Two equal-amplitude input signals that are close in frequency can appear as one on the analyzer display. When the analyzer measures a single-frequency signal, it displays the signal with the shape of the selected internal resolution bandwidth filter. As you change the filter bandwidth, you change the width of the displayed response. If you use a wide filter, two equal-amplitude input signals that are close in frequency appear as one signal. The analyzer's internal filter bandwidths determine signal resolution (how close equal-amplitude signals can be and still be distinguished).

  The resolution bandwidth function selects the internal filter bandwidth, and is defined as the 3 dB bandwidth of the filter. To resolve two signals of equal amplitude, you must set the resolution bandwidth less than or equal to the frequency separation of the two signals. If the bandwidth is equal to the separation and the video bandwidth is less than the resolution bandwidth, you will see a dip of approximately 3 dB between the peaks of the two signals.

  For swept analysis, reducing the resolution bandwidth requires an increase in sweep time to keep a measurement calibrated. For best measurement times: set the sweep time (**Sweep**, **Sweep Time**) to **Auto**, and the auto sweep time (**Sweep**, **Auto Sweep Time**) to **Norm**. Use the widest resolution bandwidth that still permits resolution of all desired signals.

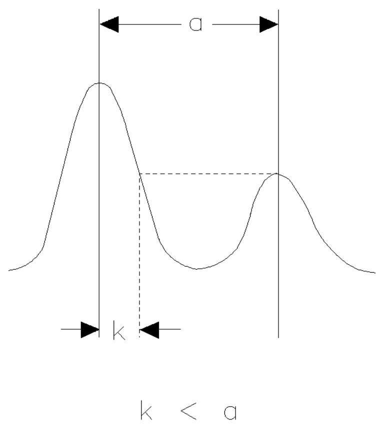
- "Finding a Small Signal Hidden by a Larger Signal" on page 34

  When signals are close together but *not* equal in amplitude, you must consider the shape of the analyzer's internal filter as well as its 3 dB bandwidth. If a small signal is too close to a larger signal, the smaller signal can be hidden by the skirt of the filter.

  To view the smaller signal, select a resolution bandwidth such that *k* is less than *a* (see Figure 4-1). The separation between the two signals (*a*) must be greater than half the filter width of the larger signal (*k*), measured at the amplitude level of the smaller signal.

  The digital filters in this instrument have filter widths about one-third as wide as typical analog RBW filters. This enables you to resolve close signals with a wider RBW (and consequently, a faster sweep).

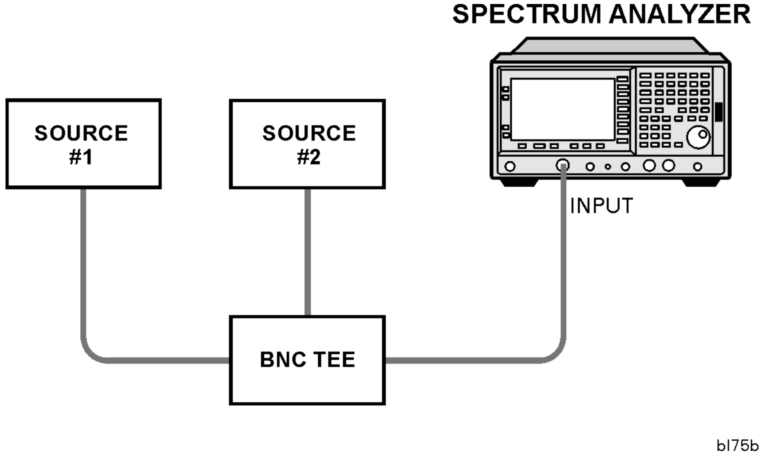
Figure 4-1 Resolution Bandwidth Requirements to Resolve Small Signals



Separating Equal-Amplitude Signals

The following example shows how to differentiate equal-amplitude signals separated by 100 kHz.

1. Connect two sources to the analyzer's RF input as follows:



2. Preset the analyzer, then set the following:

On Source 1		On Source 2	
• Frequency:	300 MHz	• Frequency:	300.1 MHz
• Amplitude:	–20 dBm	• Amplitude:	–20 dBm
• RF Output:	On	• RF Output:	On

On the Analyzer	
• Center Frequency:	300 MHz
• Span:	2 MHz
• Resolution bandwidth:	300 kHz

Press **BW/Avg**, **Resolution BW**, **3**, **0**, **0**, **kHz**.

A single signal peak should be visible.

NOTE If you cannot find the signal peak, increase the span to 20 MHz, then use signal tracking to bring the signal to the center of the screen:

Press **FREQUENCY**, **Signal Track** (press to underline **On**).

Reduce the span back to 2 MHz, then turn signal tracking off.

3. Because the resolution bandwidth must be less than or equal to the

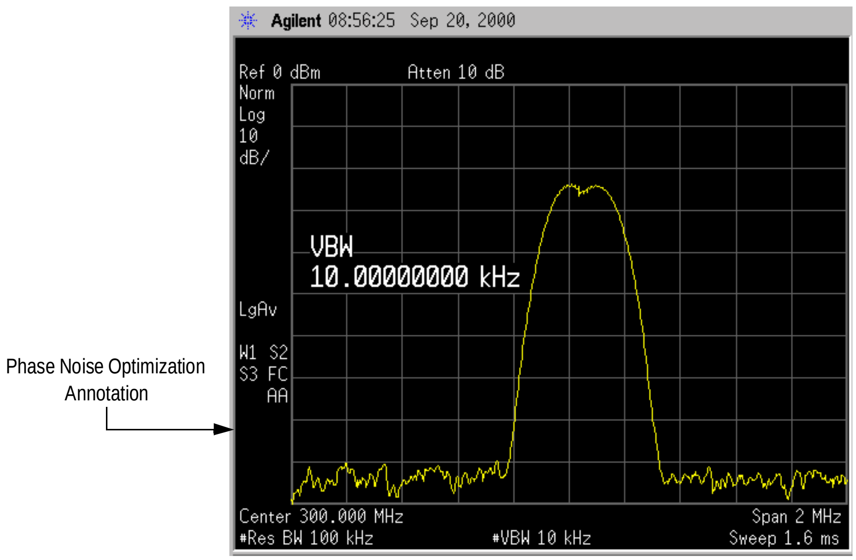
frequency separation of the two signals, change the resolution bandwidth to 100 kHz.

4. Decrease the video bandwidth to 10 kHz, as shown in Figure 4-2:

   Press **BW/Avg**, **Video BW**, **1**, **0**, **kHz**.

   Note that when you narrowed the span, the annotation for phase noise optimization changed. The optimization is now for viewing signals greater than 50 kHz away from the 300 MHz signal.

Figure 4-2 Resolving Signals of Equal Amplitude



You can experiment with reducing the resolution bandwidth further to better resolve the signals. As you reduce the resolution bandwidth, the resolution of the individual signals improves, but the sweep gets slower. For fastest measurement times, use the widest resolution bandwidth that still displays two distinct signals.

Under factory preset conditions, the resolution bandwidth is coupled (linked) to the span. Because you change the resolution bandwidth from the coupled value, a # mark appears next to `Res BW` in the lower-left corner of the screen, indicating that the resolution bandwidth is uncoupled (also see the **Auto Couple** key description in the PSA Reference Guide).

NOTE To resolve two signals of equal amplitude with a frequency separation of 200 kHz, you must use a resolution bandwidth (RBW) < 200 kHz. To enter RBW values between the 1, 3, 5 sequence provided by the up/down arrow keys, you must use the knob or data keys. In this example, the up/down arrow keys would select a 300 kHz filter which is greater than the signal separation and will not resolve the signals.

Finding a Small Signal Hidden by a Larger Signal

The following example demonstrates how to resolve two signals separated by 50 kHz and 60 dB.

1. Connect the equipment as shown on page 32, then set the sources as follows:

Source 1:	300 MHz	–20 dBm	RF output on
Source 2:	300.05 MHz	–80 dBm	RF output on

2. Preset the analyzer, then set:

- Center Frequency: 300 MHz
- Span: 300 kHz
- Resolution Bandwidth: 10 kHz

3. Set the 300 MHz signal to the reference level (top graticule):

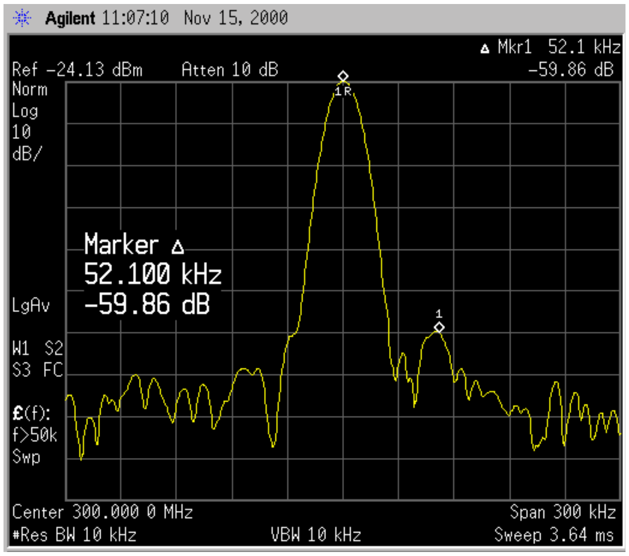
   Press **Peak Search**, **Marker →**, **Mkr → Ref Lvl**.

4. Place a marker on the smaller signal:

   Press **Marker**, **Delta**, **Peak Search**, **Next Pk Right**.

   When you use a 10 kHz filter with a typical shape factor of 4.1:1, the filter has a bandwidth of 41 kHz at the 60 dB point. Because the half-bandwidth value (20.5 kHz) is narrower than the frequency separation, the input signals are resolved, as shown in Figure 4-3.

Figure 4-3 Signal Resolution with a 10 kHz Resolution Bandwidth



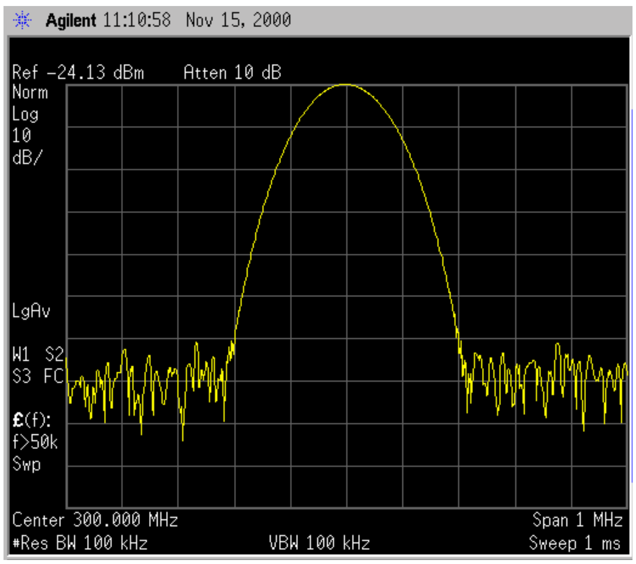
If you use a resolution bandwidth where the half-bandwidth value is wider than the frequency separation, the signals may not be resolved, as shown in Figure 4-4.

In this example, the signal amplitude difference is 60 dB. To determine the resolution capability for intermediate amplitude differences, assume the filter skirts between the 3 dB and 60 dB points are parabolic, like an ideal Gaussian filter. The resolution capability is approximately:

$$12.04\ \text{dB} \bullet \left(\frac{\Delta f}{\text{RBW}}\right)^2$$

where Δf is the separation between the signals.

Figure 4-4 Signal Resolution with a 100 kHz Resolution Bandwidth



5 Tracking a Drifting Signal

This chapter provides the following examples:

- “Tracking a Signal” on page 39

  When you measure a signal peak and must repeatedly adjust the center frequency because the signal drifts, you can use the signal track function to automatically keep the selected peak in the center of the display.

- “Measuring a Source’s Drift” on page 41

  You can use the maximum-hold function to display and hold the maximum amplitude level and frequency drift of an input signal trace. You can also use the maximum hold function to determine how much of the frequency spectrum a signal occupies.

Equipment

Both examples require a signal source.

Tracking a Signal

1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	300 MHz	• Center Frequency:	301 MHz
• Amplitude:	–20 dBm	• Span:	10 MHz
• RF Output:	On		

2. Connect the signal source to the analyzer's RF input.

   Because you set the analyzer's center frequency to a different value than that of the source's output, the 300 MHz peak is not in the center of the display.

3. Turn on signal tracking: Press **FREQUENCY**, **Signal Track** (press to underline **On**).

   This does the following:

   - Places a marker on the highest-amplitude peak.
   - Brings the selected peak to the center of the display.
   - Adjusts the center frequency each sweep to keep the selected peak in the center.
   - Turns on the signal track annotation (see Figure 5-1).

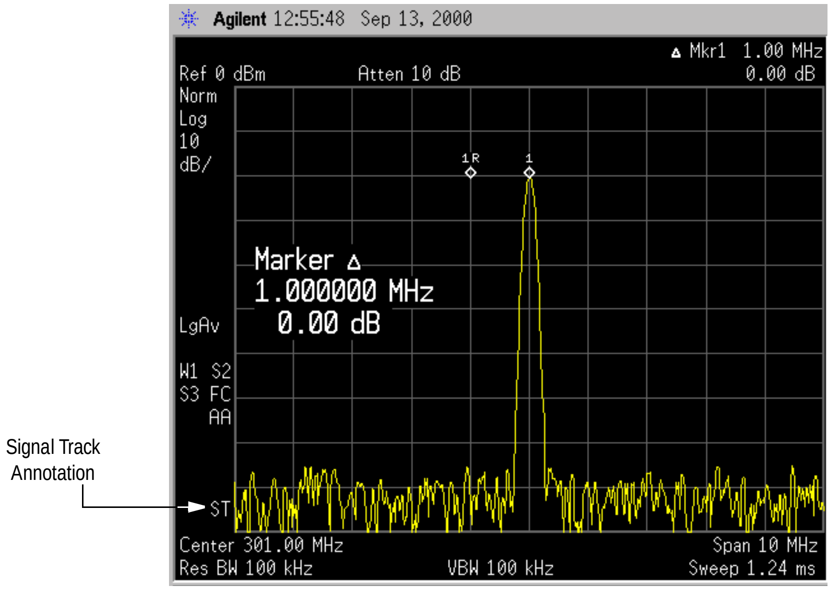
4. When you have both signal track and marker delta on, you can read any signal drift from the screen:

   Press **Marker**, **Delta**. The marker readout indicates any change in frequency and amplitude as the signal moves.

5. Slowly change the source's frequency, and note that the analyzer's center frequency changes, centering the signal with each change (see Figure 5-1).

6. Experiment with different spans, and with changing the frequency more slowly and more quickly, to see what happens.

Figure 5-1 Using Signal Tracking to Track a Drifting Signal



Measuring a Source's Drift

1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	300 MHz	• Center Frequency:	300 MHz
• Amplitude:	–20 dBm	• Span:	10 MHz
• RF Output:	On		

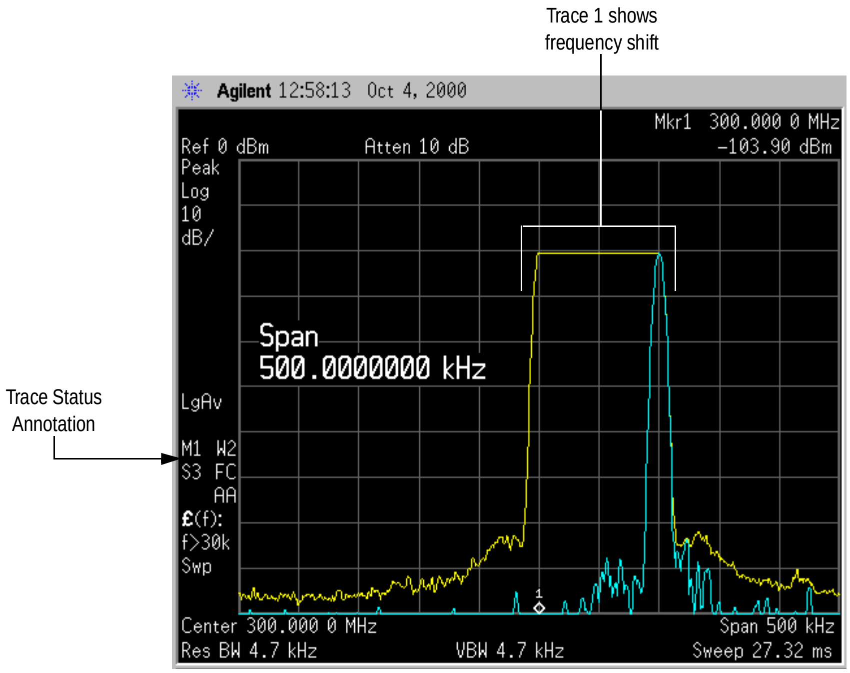
2. Connect the signal source to the analyzer's RF input, and place a marker on the peak of the signal: Press **Peak Search**.
3. Change the span to 500 kHz (if necessary, recenter the signal).
4. Measure the excursion of the signal: Press **Trace/View**, then **Max Hold**.

   As the input signal varies, maximum hold maintains the signal's maximum responses. The annotation on the left side of the screen (`M1 S2 S3`) shows that trace 1 is in maximum-hold mode; traces 2 and 3 are in store-blank mode.
5. Select trace 2: Press **Trace/View**, **Trace 1 2 3** (until 2 is underlined)
6. Clear trace 2 and have it continuously display during sweep:

   Press **Clear Write**.

   Trace 1, in maximum hold, shows any frequency shift in the signal.
7. Slowly change the source's frequency in 1 kHz steps. The analyzer display should look similar to Figure 5-2.

Figure 5-2 Viewing a Drifting Signal Using Max Hold



6 Making Distortion Measurements

This chapter provides the following examples:

- "Identifying Distortion from the Analyzer"
  - "Identifying Harmonic Distortion Products" on page 45

    High-level input signals can cause analyzer distortion products that mask input signal distortion.

  - "Measuring the Analyzer's Third-Order Intermodulation Distortion" on page 47

    Two-tone, third-order intermodulation distortion is a common test in communication systems. When two signals are present in a non-linear system (a system with components such as amplifiers and mixers), signals can interact and create distortion products close to the original signals.

- "Measuring Harmonics and Harmonic Distortion" on page 49

  This example describes how to make a harmonic measurement, and details the calculation of the total harmonic distortion for stable, modulated or unmodulated signals.

Identifying Distortion from the Analyzer

Identifying Harmonic Distortion Products

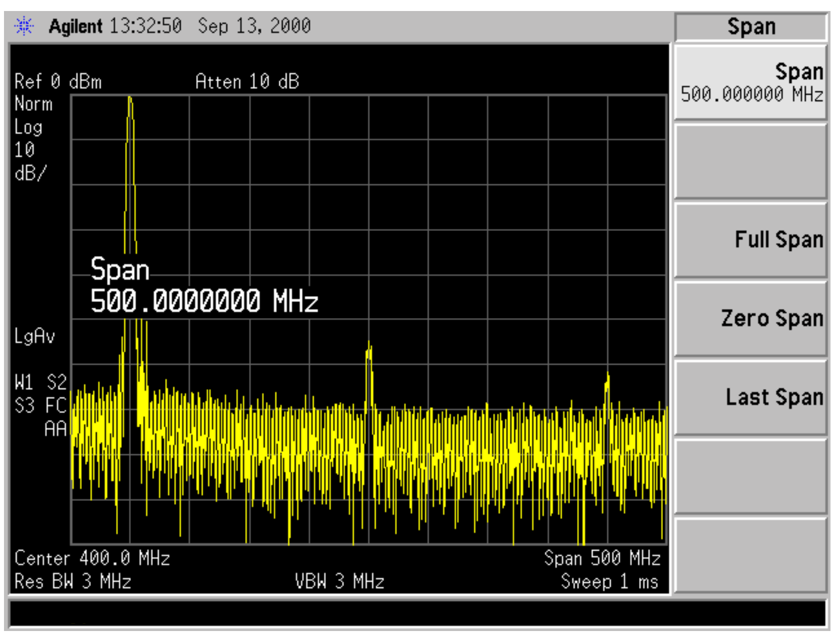
The following example uses an external signal, trace 2, and the RF attenuator to determine whether harmonic distortion products are generated by the analyzer.

1. Preset the analyzer, then set the following:

On a Signal Source		On the Analyzer	
• Frequency:	200 MHz	• Center Frequency:	400 MHz
• Amplitude:	0 dBm	• Span:	500 MHz
• RF Output:	On		

Connect the source to the analyzer. The analyzer displays the 200 MHz signal and harmonics spaced every 200 MHz (see Figure 6-1).

Figure 6-1 Harmonic Distortion



2. On the analyzer, place a marker on one of the observed harmonics, and change the center frequency to the value of that harmonic.

3. Change the span to 50 MHz.

4. Change the attenuation to 0 dB.
5. Save the screen data in trace 2:

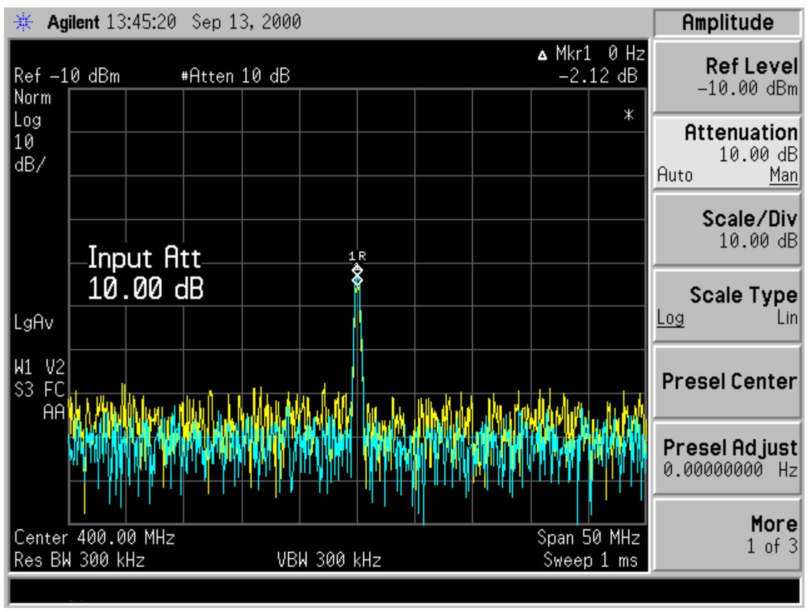
   Press **Trace/View**, **Trace 1 2 3** (to underline 2), then **Clear Write**.

   Allow the trace to update (two sweeps), then press **View**.
6. Place a delta marker on the harmonic:

   Press **Peak Search**, **Marker, Delta**.

   The analyzer display shows the stored data in trace 2 and the measured data in trace 1. The ΔMkr1 amplitude reading is the difference in amplitude between the reference and active markers.
7. Increase the RF attenuation to 10 dB. See Figure 6-2.

Figure 6-2 RF Attenuation of 10 dB



The ΔMkr1 amplitude reading comes from two sources:

- Increased input attenuation causes poorer signal-to-noise ratio. This can cause the ΔMkr1 to be positive.
- The reduced contribution of the analyzer circuits to the harmonic measurement can cause the ΔMkr1 to be negative.

Large ΔMkr1 measurements indicate significant measurement errors. For the best measurement accuracy, set the input attenuator to minimize the absolute value of ΔMkr1.

Measuring the Analyzer's Third-Order Intermodulation Distortion

The following example uses two sources at a frequency separation of 1 MHz. If you choose to use different frequencies, be sure to maintain the 1 MHz separation.

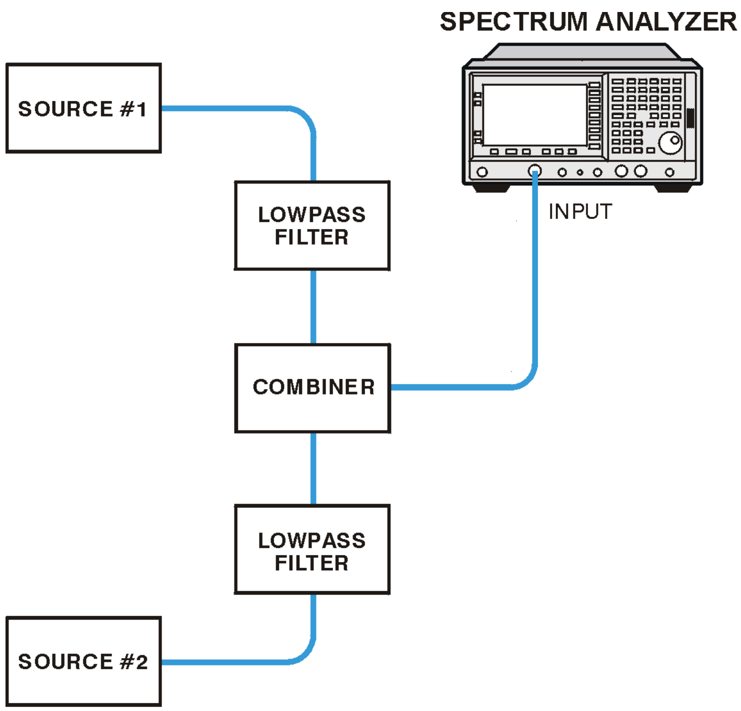
1. Set the sources for a frequency separation of 1 MHz:

Source 1:	300 MHz	–5 dBm	RF output on
Source 2:	301 MHz	–5 dBm	RF output on

2. Connect the equipment as shown in Figure 6-3, and preset the analyzer.

CAUTION Ensure that the combiner has a high degree of isolation between the two input ports so the sources do not intermodulate.

Figure 6-3 Equipment Setup



3. On the analyzer, set:

- Center Frequency: 300.5 MHz
- Span: 5 MHz (wide enough to see the distortion products)

To be sure the distortion products are resolved, adjust the resolution bandwidth as needed until the distortion products are visible.

4. Set the mixer input level to –30 dBm:

   Press **AMPLITUDE**, **More**, **More**, **Max Mixer Lvl**, **3**, **0**, –**dBm**.

5. Move the signal to the reference level:

   Press **Marker**, **Peak Search**, **Marker →**, **Mkr → Ref Lvl**.

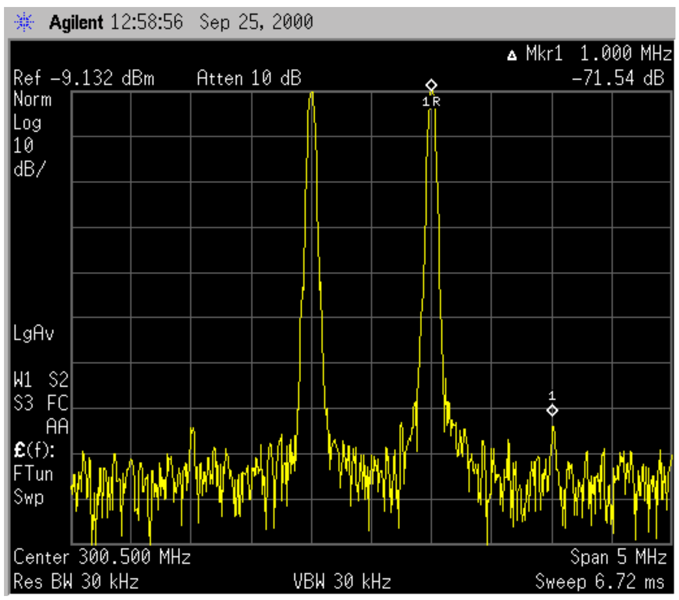
6. Reduce the resolution bandwidth until the distortion products are visible: Press **BW/Avg**, ⇓.

7. Use the delta marker function to measure the difference between the source signal and each distortion product (Figure 6-4 shows an example of this):

   Press **Marker**, **Delta**, then use the knob to move the delta marker to the distortion product you want to measure.

   For more information about measuring distortion products, see “Measuring Harmonics and Harmonic Distortion” on page 49.

Figure 6-4 Measuring a Distortion Product



Making Distortion Measurements

Measuring Harmonics and Harmonic Distortion

NOTE This measurement assumes that the highest amplitude signal displayed is the desired fundamental frequency.

The following example uses the 10 MHz Reference Output as the fundamental source, and measures harmonics and total harmonic distortion.

1. Preset the analyzer, then set the following:

- Reference Level: 10 dBm
- Center Frequency: 10 MHz
- Span: 1 MHz
- Resolution Bandwidth: 10 kHz (Press **BW/Avg**, **1**, **0**, **kHz**.)

Resolution bandwidth and attenuation are adjusted to maximize dynamic range while maintaining a reasonable sweep time. Narrower resolution bandwidths provide greater dynamic range, but lengthen sweep time. You can use the dynamic range graph (Figure 6-5 on page 50) to help determine optimal settings. In this example, harmonics are within 50 dB of the fundamental, requiring a 50 dBc dynamic range; a 10 kHz resolution bandwidth provides more than enough dynamic range to view the second harmonic.

When measuring the *N*th harmonic, the analyzer uses the narrowest resolution bandwidth that is *N* times the resolution bandwidth used to measure the fundamental. Widening the resolution bandwidth enables the measurement to capture all modulation on the harmonics. An asterisk (*) appears next to the amplitudes of measured harmonics for which the desired resolution bandwidth cannot be set. As long as the signal at the harmonic has less modulation width than the RBW, the measurement is accurate.

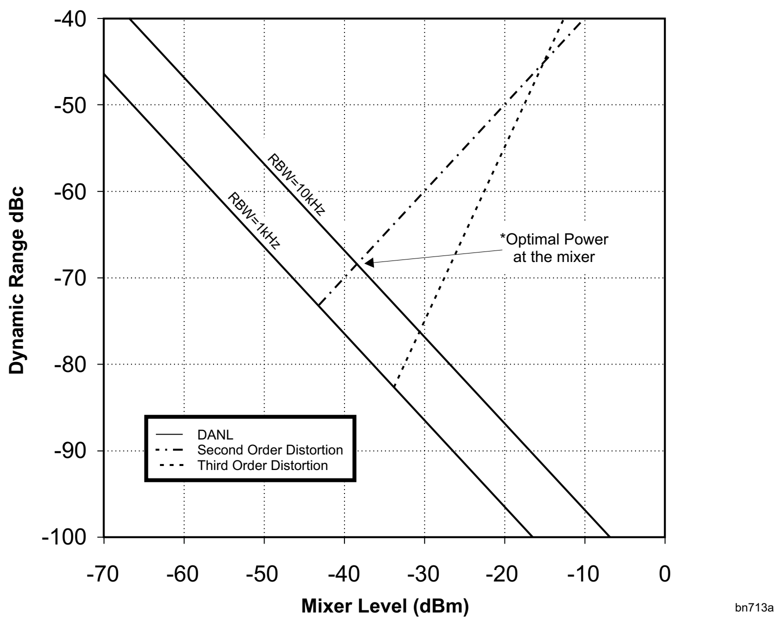
- Attenuation: 40 dBm (Press **AMPLITUDE**, **Attenuation**, **4**, **0**, **dB**.)

Attenuation is set for optimal power at the mixer, which occurs at the intercept of the second order harmonic line and the Displayed Average Noise Level (DANL) line for the resolution bandwidth selected (see the note inside Figure 6-5). This occurs at a mixer level of approximately –29 dBm. The input level from the 10 MHz Reference Output is +5 dBm in this example. Using the mixer level and the input level in the equation below provides us with an optimal attenuation setting of 34 dB.

Attenuation Setting (dB) = Input Level (dBm) – Mixer Level

2. With the rear panel 10 MHz output on (As described on page 16, in Step 2.), connect the analyzer's rear panel `10 MHz OUT (SWITCHED)` to the front-panel RF input.

Figure 6-5 Dynamic Range Graph



3. To calculate the total harmonic distortion of a signal, perform the following steps, in the following order:

   a. Determine the frequencies of the harmonics.

   b. For each harmonic:

      1. Select the harmonic: Press **Marker**, then use the knob to move the marker to the desired harmonic.

      2. Span down to zero span: Press **Span**, **Zero Span**.

      3. Measure the amplitude.

NOTE To display the amplitude in voltage units: press **Amplitude**, **More**, **Y-Axis Units**, **Volts**.

   c. Divide the root-sum-squares of the harmonic voltages by the fundamental signal voltage. Then multiply the results by 100 to arrive at a percentage:

$$\%\mathrm{THD} = 100 \times \frac{\left(\sqrt{\sum_{h=2}^{h_{max}} {E_h}^2}\right)}{E_f}$$

   where:

   %THD = Total Harmonic Distortion as a percentage
   h = harmonic number

H_{max} = Maximum Harmonic Value listed
E_h = voltage of harmonic h
E_f = voltage of fundamental signal

Example THD Calculation

Number of harmonics (Hmax) = 5; measured values are:

$$E_f = 5\text{ dBm} = 3.162\text{ mW} = 397.6\text{ mV}$$
$$E_2 = -42\text{ dBc} = -37\text{ dBm} = 199.5\text{ nW} = 3.159\text{ mV}$$
$$E_3 = -26\text{ dBc} = -21\text{ dBm} = 7.943\ \mu\text{W} = 19.93\text{ mV}$$
$$E_4 = -49\text{ dBc} = -44\text{ dBm} = 39.81\text{ nW} = 1.411\text{ mV}$$
$$E_5 = -36\text{ dBc} = -31\text{ dBm} = 794.3\text{ nW} = 6.302\text{ mV}$$

then,

$$\text{THD} = 100 \times \frac{\sqrt{3.159\text{ mV}^2 + 19.93\text{ mV}^2 + 1.411\text{ mV}^2 + 6.301\text{ mV}^2}}{397.6\text{ mV}} = 5.33\%$$

NOTE Alternate Method

You can use the analyzer's built-in harmonic distortion measurement capability: Press **Measure**, **Harmonic Distortion**, **Trace/View**, **Harmonics & THD**.

7 Measuring Noise Signals

There are several ways to measure noise power. This chapter provides the following examples:

- "Measuring Noise at a Single Frequency" on page 55

  This example uses the marker noise function. In this example, you must pay attention to the potential errors due to a discrete signal (spectral components). This measurement uses the analyzer's 50 MHz reference signal.

- "Measuring Signal-to-Noise Levels" on page 57

  For this measurement, the signal (carrier) is a discrete tone (the 50 MHz amplitude reference signal).

  If the signal is a carrier that is modulated under normal operation, you can use the amplitude reference signal as the signal of interest and the noise of the analyzer for the noise measurement. In this example, however, you set the input attenuator such that both the signal and the noise are well within the calibrated region of the display.

- "Measuring Total Noise Power" on page 58

  This example uses markers to set the frequency span over which you measure power. Markers enable you to select and measure any portion of the displayed signal. The analyzer sets the sample display detection mode, but you must set all other parameters.

Measuring Noise at a Single Frequency

This example uses the analyzer's 50 MHz reference signal, and the analyzer's marker noise function.

1. With nothing connected to the RF input, preset the analyzer and set:

   - Attenuation 40 dB
   - Center Frequency: 49.98 MHz
   - Span: 100 kHz

2. Turn on the analyzer's 50 MHz amplitude reference signal:

   Press **Input/Output**, **Input Port**, **Amptd Ref (f=50MHz)**.

3. Activate the noise marker: Press **Mkr Fctn**, **Marker Noise**.

   Note that the display detection changes to Avg; the marker floats between the maximum and the minimum noise. The marker readout is in dBm(1Hz) or dBm per unit bandwidth (see Figure 7-1 on page 56).

   For noise power in a different bandwidth, add $10 \times \log(\mathrm{BW})$. For example, for noise power in a 1 kHz bandwidth, add 30 dB ($10 \times \log(1000)$) to the noise marker value.

4. To reduce the variations of the sweep-to-sweep marker value, change the sweep time to 3 seconds: Press **Sweep**, **Sweep Time, 3**, **s**.

NOTE Noise measurements are noisy. Increasing the sweep time enables the average detector to average over a longer time interval, thus reducing the variations in the results.

5. The noise marker value is based on the mean of 33 trace points centered at the marker. With a total of 601 points across the entire trace, the 33 points cover approximately half of a division.

   To see the effect, press **Marker** and use the knob to move the marker to the 50 MHz signal.

   The marker does not go to the peak of the signal because not all 33 trace points are at the peak of the signal.

6. Widen the resolution bandwidth to 10 kHz: Press **BW/Avg**, **1**, **0**, **kHz**.

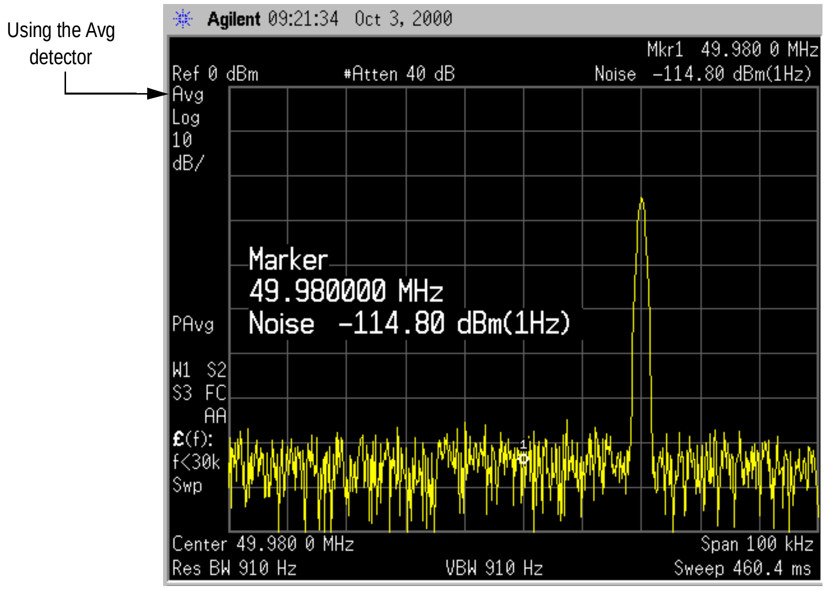
7. Again press **Marker** and move the marker to the signal.

   The 33 trace points still cover over 0.55 divisions, but the signal level is close to constant over this range, so the marker is closer to the peak of the signal.

8. Return the resolution bandwidth to automatic mode:

   Press **BW/Avg**,**Res BW** (until **Auto** is underlined).

Figure 7-1 Activating the Noise Marker



9. Press **Marker** and use the knob to place the marker at 49.99625 MHz to measure the noise very close to the signal.

   Note that the marker reads an incorrect value, because some of the trace points are on the skirt of the signal response.

10. Set the analyzer for zero span: Press **SPAN**, **Zero Span**, **Marker**.

    Note that the marker value is now correct.

Measuring Signal-to-Noise Levels

This example uses the analyzer's 50 MHz amplitude reference signal.

1. Preset the analyzer, then set:

- Reference Level: –10 dBm
- Attenuation 40 dB
- Center Frequency: 50 MHz
- Span: 1 MHz

2. Turn on the analyzer's 50 MHz amplitude reference signal, as described on page 55, in Step 2.

3. Place a marker on the peak of the signal, then place a delta marker in the noise at a 200 kHz offset: Press **Marker**, **Delta**, ⇑, ⇑, **kHz**.

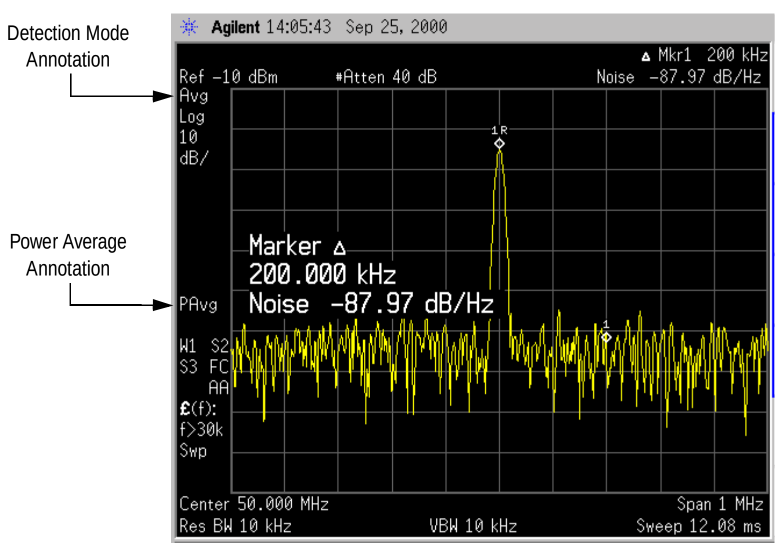
4. Turn on the marker noise function: Press **Mkr Fctn**, **Marker Noise**. This lets you view the results of the signal-to-noise measurement (Figure 7-2).

   Read the signal-to-noise in dB/Hz, which is the noise value determined for a 1-Hz noise bandwidth. For noise value at a different bandwidth, increase the ratio by $10 \times \log(BW)$. For example, if the analyzer reads –70 dB/Hz, but you used a channel bandwidth of 30 kHz: $S/N = -70dB/Hz + 10 \times \log(30kHz) = -25.2dB/30kHz$

   Note that the detection mode is now Avg, and that the power average (PAvg) display function is selected.

NOTE If the delta marker is within one-half a division of the response to a discrete signal (in this case, the amplitude reference signal), there is potential for measurement error.

Figure 7-2 Measuring the Signal-to-Noise



Measuring Total Noise Power

You can use markers to set the frequency span over which you measure power. Markers enable you to select and measure any portion of the displayed signal. The analyzer selects the average display detector, but you must set all other parameters.

1. Preset the analyzer, then set:

•	Reference Level:	–20 dBm	•	Center Frequency:	50 MHz
•	Attenuation	40 dB	•	Span:	100 kHz

2. Set the marker span to 40 kHz:

   Press **Marker**, **Span Pair** (until **Span** is underlined), **4**, **0**, **kHz**.

NOTE

Alternate Method

You can also use **Delta Pair** to set the measurement start and stop points independently (as described on page 18).

The resolution bandwidth should be about 1 to 3% of the measurement (marker) span (which is 40 kHz in this example). The analyzer's default resolution bandwidth is approximately 1 kHz.

3. Measure the power between markers:

   Press **Mkr Fctn**, **Band/Intvl Power**.

   The analyzer displays the total power between the markers, as shown in Figure 7-3 on page 59.

4. Add a discrete tone (the analyzer's 50 MHz amplitude reference signal) to see how it affects the reading (also see Figure 7-4 on page 59):

   Press **Input/Output**, **Input Port**, **Amptd Ref Out (f=50MHz)**.

5. Move the measured span:

   Press **Marker**, **Span Pair** (**Center** underlined).

   Then use the knob to exclude the tone and note reading.

Figure 7-3 Viewing Power Between the Markers

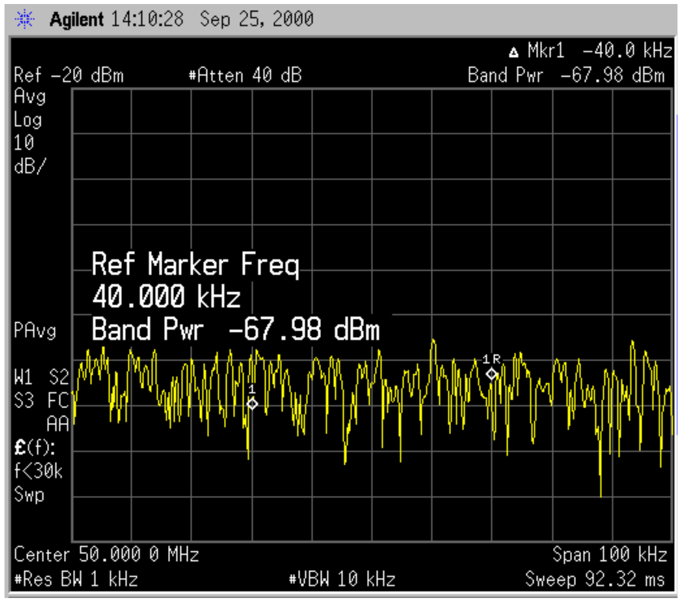
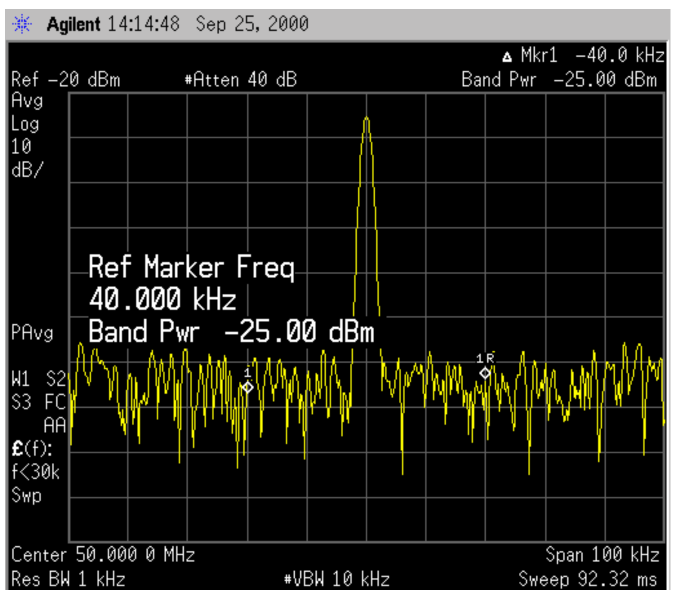


Figure 7-4 Viewing the 50 MHz Signal Between the Markers



8 Measuring the Power of Digital Signals

There are several ways to measure the power of noise, or of the noise-like signals which are common in digitally modulated systems. This chapter provides the following examples:

- "Making Power Measurements on Burst Signals" on page 63

  The Burst Power measurement is a very accurate method of determining the average power for the specified burst. The analyzer is set into zero-span mode, with a sweep time that captures at least one burst. The default is just more than a single burst, but the user may change this using the 'Sweep Time' softkey in the 'Sweep' menu.

- "Making Statistical Power Measurements (CCDF)" on page 67

  The CCDF (complimentary cumulative distribution function) measurement is a statistical measurement of a signal's high-level or peak power. It is a graphical representation of the percentage of time a signal exceeds its average power, and by how much this average is exceeded.

  All CDMA signals, and W-CDMA signals in particular, are characterized by high power peaks that only occur occasionally. It is important that these peaks are preserved, otherwise individual data channels will not be received properly. A signal with higher probabilities of high peaks is often more distorted by signal processing elements that cannot handle the peaks. If a CDMA system works well most of the time, only failing occasionally, the cause can often be traced to compression of the higher peak signals.

- "Making Measurements of Adjacent Channel Power (ACP)" on page 70

  ACP measures the total power in the specified channel and its adjacent channels for up to six pairs of offset frequencies. The offset frequencies can be modified at any time, but the default values are those specified by the relevant international standard that you select. The results are displayed by default both as power relative to the carrier (in dBc) and as absolute power (dBm).

- "Making Measurements of Multi-Carrier Power (MCP)" on page 74

  MCP measures the total power in two or more transmit channels and their adjacent channels for up to three pairs of offset frequencies. The offset frequencies can be modified at any time, but the default values are those specified by the relevant international standard that you select. This measurement is available with no radio standard selected or when you select any of the following radio standards: IS-95, J-STD-008, all cdma2000 standards, or W-CDMA. Results for carriers without power present are displayed relative to the reference carrier. Results for adjacent channels are displayed both in absolute power (dBm) and as power relative to the reference carrier (dBc).

Making Power Measurements on Burst Signals

The following example demonstrates how to make a burst power measurement on a Bluetooth signal broadcasting at 2.402 GHz.

1. Connect a DH1 Bluetooth signal to the analyzer input, preset the analyzer and set:

• Mode:	Spectrum Analysis
• Mode setup, radio standard:	Bluetooth
• Mode setup, std setup, Packet Type:	DH1
• Center Frequency:	2.402 GHz

   Note that burst signal levels > –5 dBm may overload the analyzer. You may need to set input attenuation to auto so the required attenuation is added.

2. Select the burst power measurement.

   Press **MEASURE, More, Burst Power**.

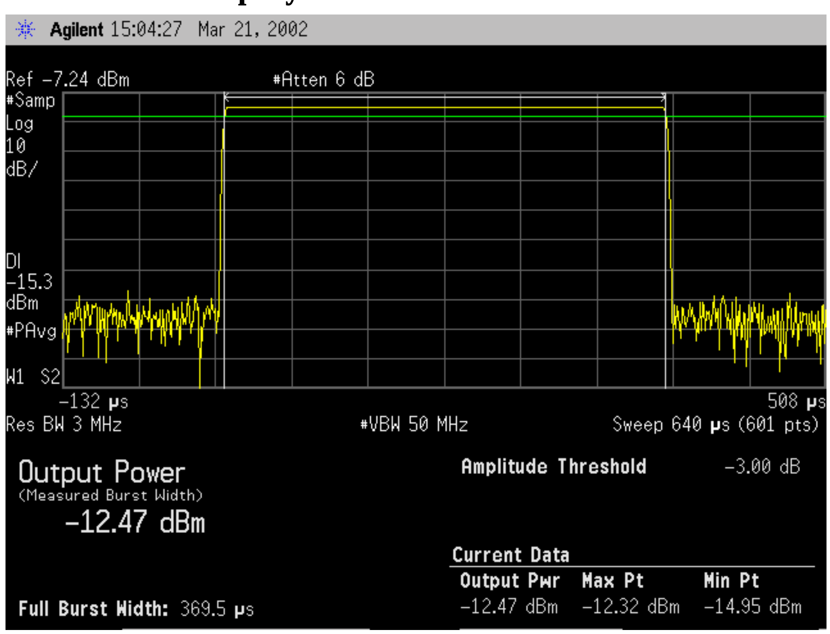
3. Set the best reference level for this measurement on this signal.

   Press **Meas Setup, Optimize Ref Level**.

4. View the results using the full screen.

   Press **Display, Full Screen** and you should see results similar to Figure 8-1.

Figure 8-1 Full Screen Display of Burst Power Measurement Results



NOTE Alternate Methods

1. If an external trigger is available, connect this to Trigger In on the rear of the instrument and press **Trig**, **Ext Rear**, or connect to Ext Trigger Input on the front panel and press **Trig**, **Ext Front**.

2. You could also select Video trigger. It might then be necessary to adjust the trigger level (as indicated by the horizontal green line) by rotating the front panel knob or by entering a numeric value on the keypad. For this example, set the trigger level to -30 dBm.

NOTE Although the Trigger Level allows the analyzer to detect the presence of a burst, the Burst Power measurement is determined by the threshold level, as described next.

5. Set the relative level above which the burst power measurement will be calculated.

   Press **Meas Setup**, **Threshold Lvl** (Rel) **-10**, **dB**.

   The mean power of the burst is measured from the point where the rising signal level rises above the threshold (green line) to the point where the signal passes below it. In this example, the threshold level has been set to be 10 dB below the peak value. Refer to Figure 8-2.

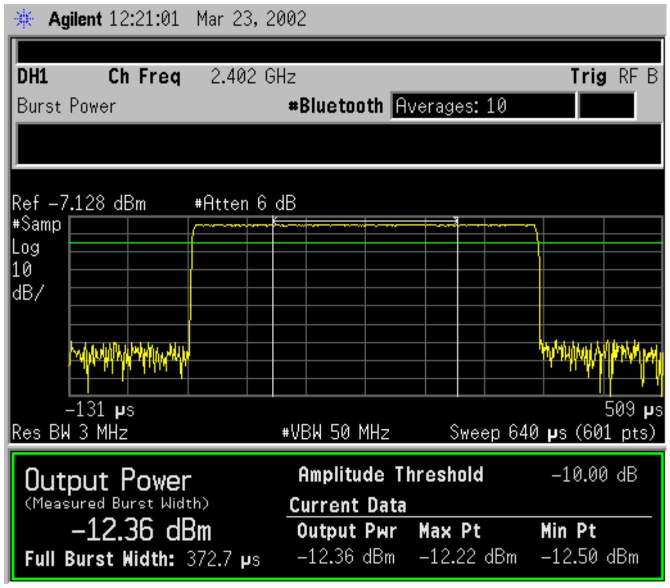
6. To specify the burst width for which the measurement will be taken: Press **Meas Setup**, **Meas Method**, **Measured Burst Width**, **Burst Width** (Man), **200, µs**. This will measure just the central 200 µs of the burst.

   The burst width is indicated on the screen by two vertical white lines as shown in Figure 8-2. Manually setting the burst width allows you to make it a long time interval (to include the rising and falling edges of the burst) or to make it a short time interval, thus measuring only a small central section of the burst.

NOTE The Bluetooth standard states that power measurements should be taken from the central 60% of the burst only. Other radio standards use different figures.

NOTE If you set the burst width manually to be wider than the screen's display, the vertical white lines will move off the edges of the screen. This could give misleading results as only the data on the screen can be measured.

Figure 8-2 Manually Setting the Burst Width

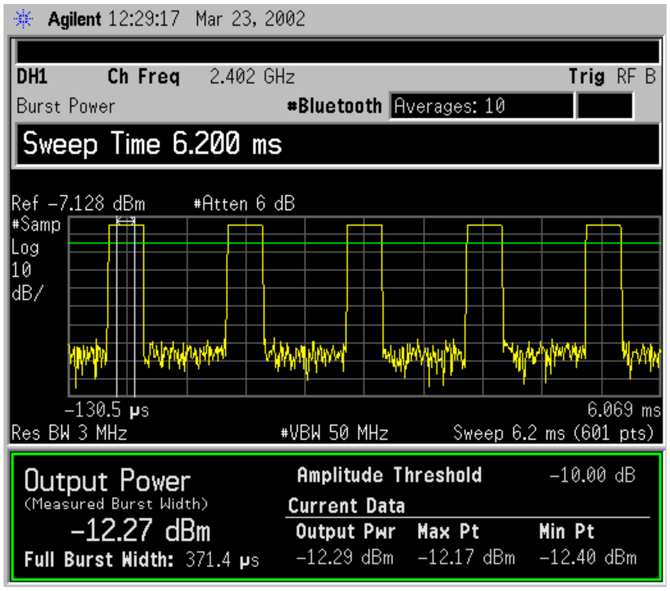


7. Change the sweep time to display more than one burst at a time.

   Press **Sweep**, **Sweep Time**, **6200**, **μs** (or **6.2**, **ms**).

   The screen display will now show several bursts in a single sweep as shown in Figure 8-3 below. The burst power measurement will measure the mean power of the first burst, indicated by the vertical white lines either around it or, as in this example, within it.

Figure 8-3 Displaying Multiple Bursts



NOTE Although the burst power measurement still runs correctly when several bursts are displayed simultaneously, the timing accuracy of the measurement is degraded. For the best results (including the best trade-off between measurement variations and averaging time), it is recommended that the measurement be performed on a single burst.

Making Statistical Power Measurements (CCDF)

The following example shows how to make a CCDF measurement on a W-CDMA signal broadcasting at 1.96 GHz.

1. Connect a W-CDMA signal to the analyzer input, preset the analyzer and set:

• Mode:	Spectrum Analysis
• Mode setup, radio standard:	3GPP W-CDMA
• Mode setup, std setup, Device:	BTS
• Center Frequency:	1.96 GHz

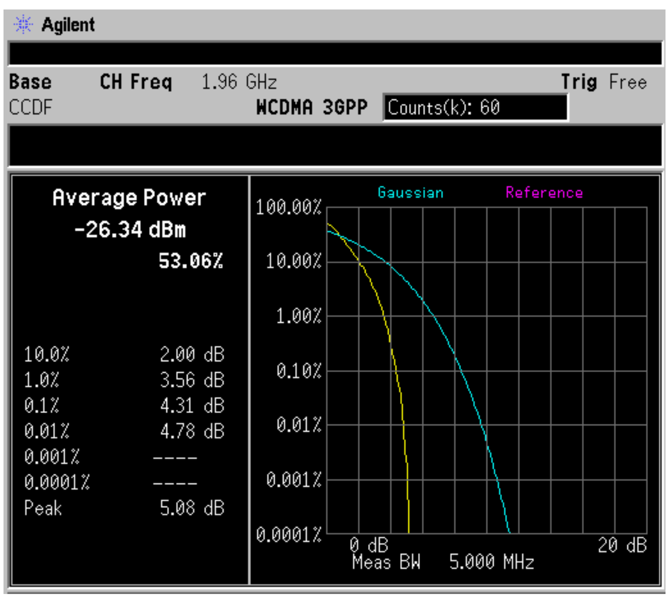
2. Select the power statistics (CCDF) measurement

   Press **MEASURE, Power Stat CCDF**.

3. Set the best attenuation and reference level for this measurement on this signal.

   Press **Meas Setup**, **Optimize Ref Level**.

Figure 8-4 Power Stat CCDF Measurement on a W-CDMA Signal



4. Store your current measurement trace for future reference.

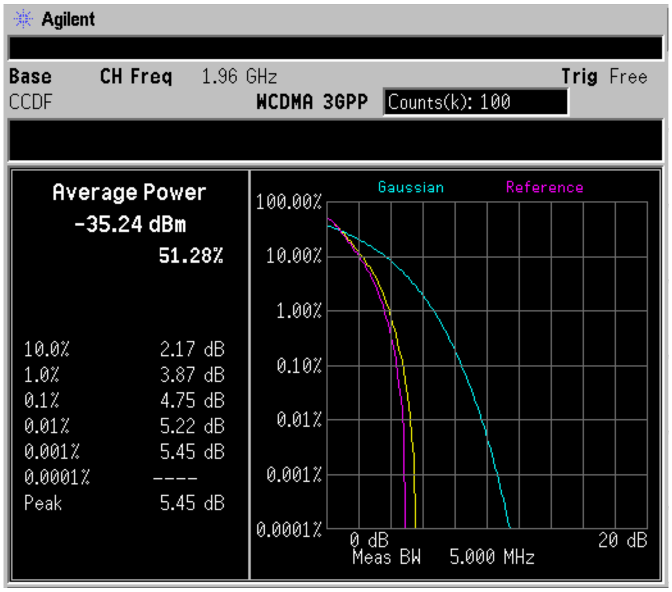
   Press **Display**, **Store Ref Trace**.

   When the Power Stat CCDF measurement is first made, the graphical display should show a signal typical of pure noise. This is labelled ‘Gaussian’, and is shown in aqua. Your measurement will show as a yellow plot. You have stored this measurement plot for easy comparison with subsequent measurements.

5. Display the stored trace.

   Press **Display**, **Ref Trace** (On). The stored trace from your last measurement is displayed as a magenta plot (as shown in Figure 8-5), and allows direct comparison with your current measurement.

Figure 8-5 Storing and Displaying a Power Stat CCDF Measurement



6. Change the measurement bandwidth to 1 MHz.

   Press **Meas Setup, Meas BW**, **1**, **MHz**.

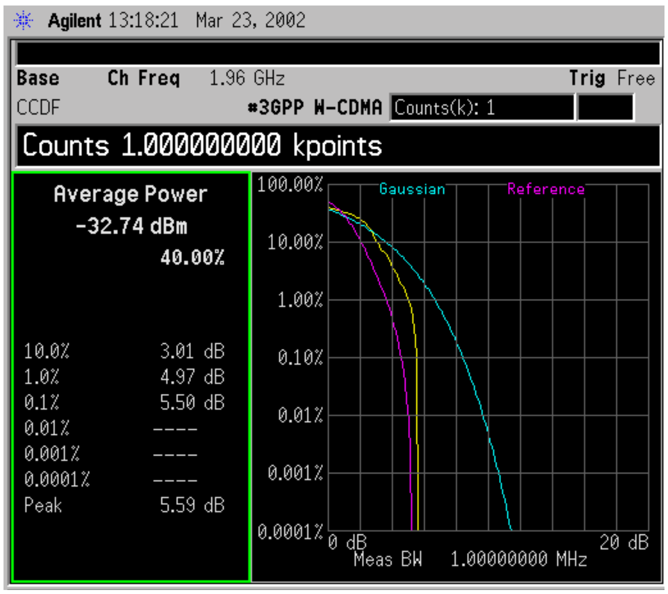
NOTE If you choose a measurement bandwidth setting that the instrument cannot display, it will automatically set itself to the closest available bandwidth setting.

7. Change the number of measured points from 100,000 (100k) to 1,000 (1k).

   Press **Meas Setup, Counts**, **1 kpoints**. Reducing the number of points decreases the measurement time, however the number of points is a factor in determining measurement uncertainty and repeatability. Notice how the displayed plot loses a lot of its smoothness. You are gaining speed but reducing repeatability and increasing measurement uncertainty.

NOTE The number of plots collected per sweep is dependent on the sampling rate and the measurement interval. The number of samples that have been processed will be indicated at the top of the screen. The graphical plot will also be updated so you will be able to see it getting smoother as measurement uncertainty is reduced and repeatability improves.

Figure 8-6 Reducing the Number of Measurement Points to 1,000



8. Change the scale of the X-axis to optimize your particular measurement.

   Under **Span X Scale, Scale/Div, 1**, **dB**.

Making Measurements of Adjacent Channel Power (ACP)

The following example shows how to make an ACP measurement on a W-CDMA Base Station signal broadcasting at 1.96 GHz.

1. Connect a W-CDMA signal to the analyzer input, preset the analyzer and set:

   - Mode: Spectrum Analysis
   - Mode setup, radio standard: 3GPP W-CDMA
   - Mode setup, std setup: BTS
   - Center Frequency: 1.96 GHz

2. Select the Adjacent Channel Power measurement.

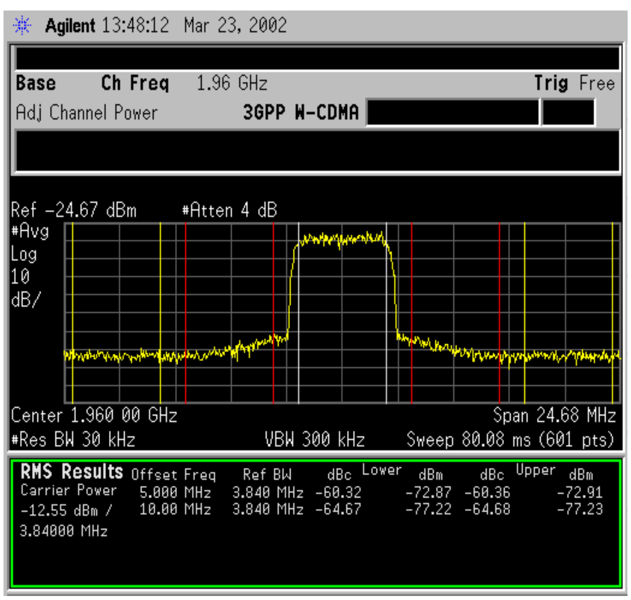
   Press **MEASURE, ACP**.

3. Set the optimum signal reference level for this measurement.

   Press **Meas Setup**, **Optimize Ref Level**. Your screen should now look like Figure 8-7.

NOTE This optimization protects against input signal overloads, but does not necessarily set the input attenuation for optimum measurement dynamic range.

Figure 8-7 ACP Measurement on a Base Station W-CDMA Signal



The Frequency Offsets, Channel Integration Bandwidths, and Span settings can all be modified. They default to the relevant settings for the radio standard you have currently selected.

Two vertical white lines indicate the bandwidth limits of the central channel being measured.

Offsets A and B are designated by the adjacent pairs of red and yellow lines, in this case: 5 MHz and 10 MHz from the center frequency respectively.

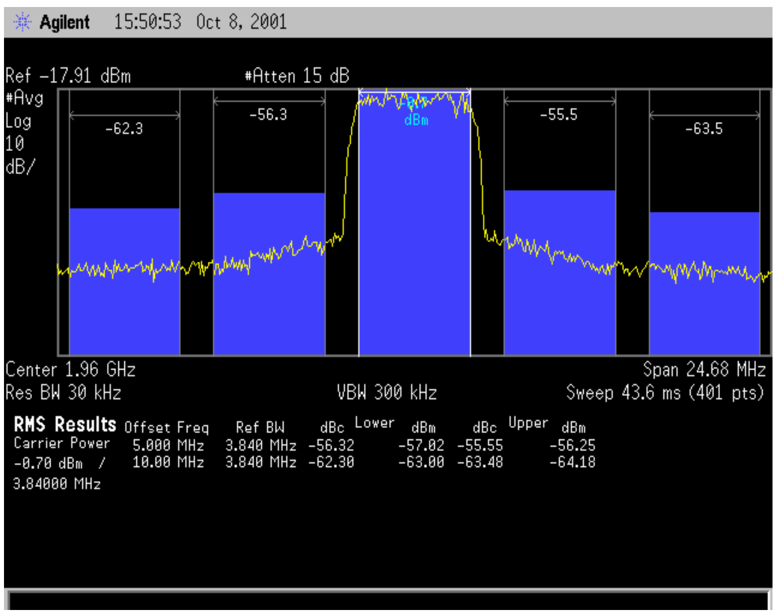
4. Select the combined spectrum and bar graph view of the results.

   Press **Trace/View, Combined**.

5. View the results using the full screen.

   Press **Display**, **Full Screen** to display a larger view of the trace as shown in Figure 8-8.

Figure 8-8 ACP Measurement in Full Screen Display

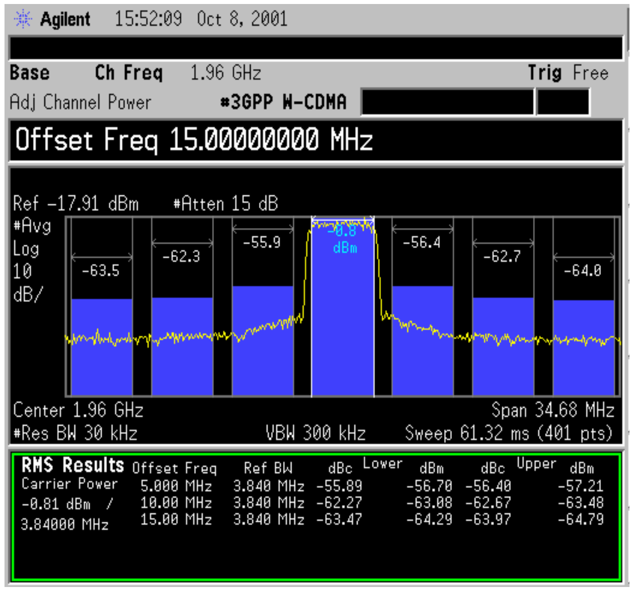


6. Define a new offset.

   Press **Meas Setup**, **Offset/Limits**, **Offset**, **C**, **Offset Freq (On)**, **15**, **MHz** to set a third pair of offset frequencies.

   This third pair of offset frequencies will be offset by 15.0 MHz from the center frequency and are shown on the screen as the third blue bar graph from the central channel. An example screen with this extra pair of frequencies is shown in Figure 8-9. Three further pairs of offset frequencies (D, E and F) are available and are displayed similarly.

Figure 8-9 Measuring a Third Adjacent Channel



7. Set pass/fail limits for each offset.

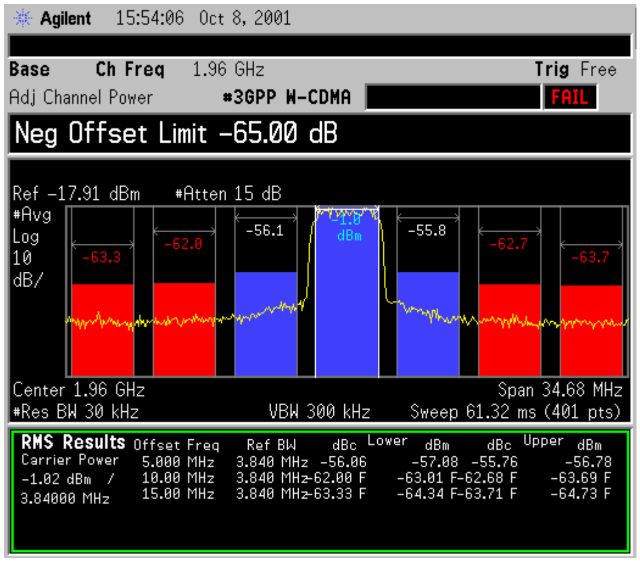
   Press **Offset (A)**, **Neg Offset Limit**, **–55 dB**, **Pos Offset Limit**, **–55 dB**, **Offset (B)**, **Neg Offset Limit**, **–65 dB**, **Pos Offset Limit**, **–65 dB**, **Offset (C)**, **Neg Offset Limit**, **–65 dB**, **Pos Offset Limit**, **–65 dB**.

8. Turn the limit test on.

   Press **Meas Setup**, **More, Limit Test** (press until **On** is underlined) to show the results as in Figure 8-10.

   Offset A has passed, however Offsets B and C have failed. Failures are identified by the red letter "F" next to the levels (dBc and dBm) listed in the lower portion of the window called, "RMS Results". The offset bar graph is also shaded red to identify a failure.

Figure 8-10 Setting Offset Limits



NOTE You may increase the repeatability by increasing the sweep time.

Making Measurements of Multi-Carrier Power (MCP)

The following example shows how to make an MCP measurement on W-CDMA Base Station broadcasting 10 carriers. Eight carriers have power present at the following frequencies:1.0225 GHz, 1.0175 GHz, 1.0125 GHz, 1.0075 GHz, 992.5 MHz, 987.5 MHz, 982.5 MHz, and 992.5 MHz. This measurement is available with no radio standard selected or with any of the following radio standards: IS-95, J-STD-008, all cdma2000 standards, or W-CDMA.

NOTE When **Radio Std**, **None** is selected you must manually set most parameters required to perform this measurement. When selecting **Radio Std**, **W-CDMA 3GPP**, these parameters are already set by the analyzer.

1. Connect a W-CDMA signal to the analyzer input, preset the analyzer and set:

    - Mode: Spectrum Analysis
    - Mode setup, radio standard: 3GPP W-CDMA
    - Mode setup, std setup, Device: BTS
    - Center Frequency: 1.0 GHz

2. Select the Multi-Carrier Power measurement.

   Press **MEASURE, Multi-Carrier Power**.

3. Set the optimum signal reference level for this measurement.

   Press **Meas Setup**, **Optimize Ref Level**.

4. Set the carrier number to 10.

   Press **Carrier Setup**, **Carriers, 1, 0, Enter.**

5. Configure carrier 5 to have no power present.

   Press **Configure Carriers**, **Carrier, 5, Enter, Carrier Pwr Present,** (Press to underline **No**).

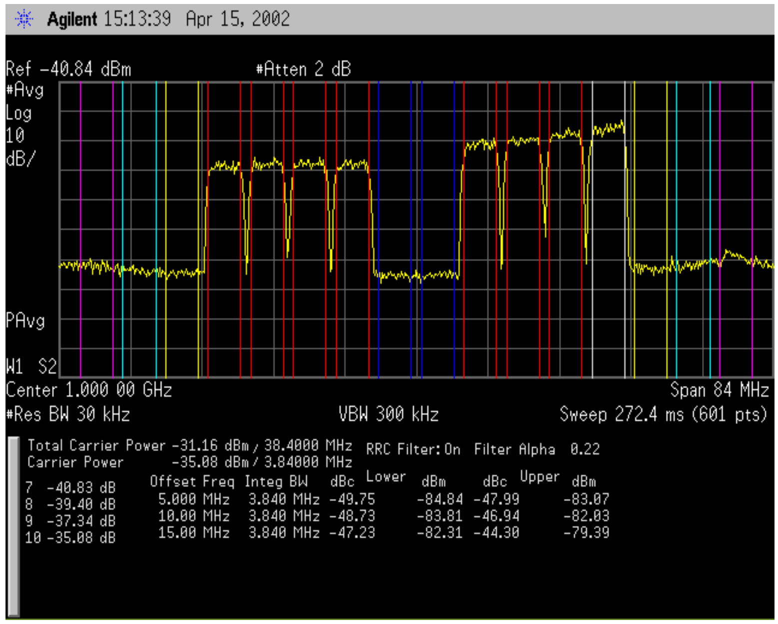
6. Repeat step 5, configuring carrier 6 to have no power present.

7. Display the results in full screen view. Refer to Figure 8-11.

   Press **Display**, **Full Screen**.

Measuring the Power of Digital Signals

Figure 8-11 MCP Measurement on 10 Base Station W-CDMA Carriers



In this example, the intermodulation falls outside the transmit channels which are marked by the colored vertical lines. The white set indicates the reference carrier. The red sets contain the carriers with power present and the blue lines mark the carriers without power present. Limits for the upper and lower offsets can also be set as shown in the example: "Making Measurements of Adjacent Channel Power (ACP)" on page 70.

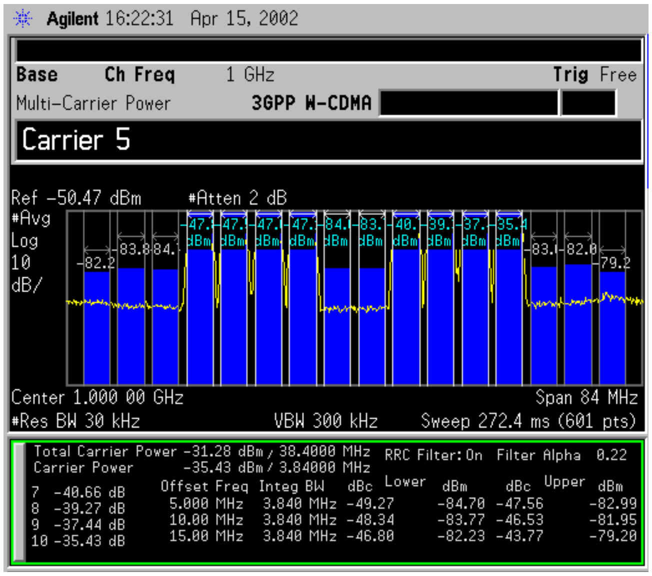
8. View the results table of carriers 7-10.

   Press **Meas Setup**, **Carrier Result, 7, Enter**.

9. View the results in a combined spectrum and bar graph. Refer to Figure 8-12.

   Press **Trace/View, Combined**.

Figure 8-12 Combined Spectrum and Bar Graph View



10.Save the results file to a disk.

Press **File**, **Save**, **Type**, **Measurement Results**, **Save Now**. The results are stored in a comma separated values format to be viewed by any personal computer spreadsheet application. All data shown on the display is included in this file.

9 Using External Millimeter Mixers (Option AYZ)

External millimeter mixers can be used to extend the frequency range of the spectrum analyzer. Agilent Technologies manufactures external mixers that do not require biasing and cover frequency ranges from 18 GHz to 110 GHz. Other manufacturers sell mixers that extend the range to 325 GHz, but may require biasing. The spectrum analyzer supports both types of mixers.

This chapter provides the following examples:

- "Using Unpreselected Millimeter-wave Mixers" on page 79

  When using unpreselected mixers, multiple mixing products will be shown on the analyzer display. Signal identification is required to interpret these results correctly.

  The output of a harmonic mixer will contain the sum and difference of the input signal with the LO and all of its harmonics. A signal is displayed on the analyzer whenever the input signal and the harmonic frequency of the LO differ by the defined IF (intermediate frequency). As a result, within a single harmonic band, a single input signal can produce multiple responses on the analyzer display, only one of which is valid for the current displayed frequency range.

  These responses come in pairs, where members of the valid response pair are separated by 642.8 MHz. When the analyzer is using negative mixing harmonics the right-most signal in the pair is the correct response. For positive harmonics the left-most signal of the pair is the correct response.

- "Using Preselected Millimeter-Wave Mixers" on page 83

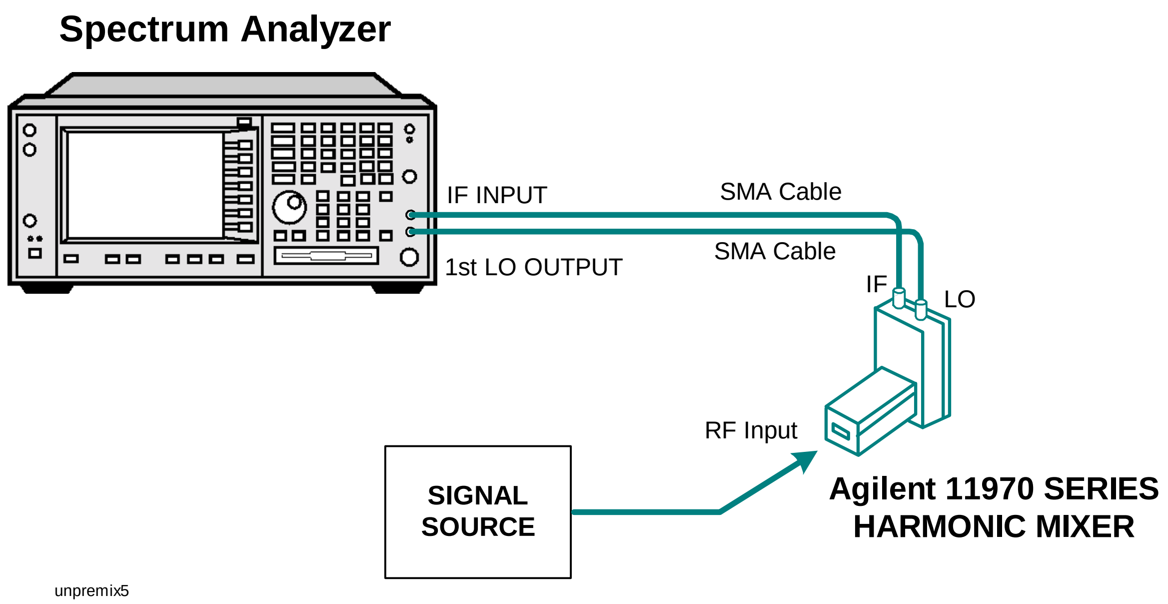
  Preselected mixers apply a tracking filter to the input signal before sending it to the mixer. This makes the displayed results easier to understand because it eliminates the multiple mixing products that are displayed using unpreselected mixers.

  You must align the frequency of the preselector filter to the tuned frequency of the analyzer before making any measurements. This must be done whenever the mixer is connected to a different analyzer. The alignment should be checked periodically.

Using Unpreselected Millimeter-wave Mixers

1. Connect the signal source and harmonic mixer to the analyzer as shown. In this example we will use an A Band harmonic mixer.

Figure 9-1 Setup with Unpreselected External Mixers



CAUTION The analyzer local oscillator output power is approximately. Be sure that your external harmonic mixer can accommodate the power level before connecting it to the analyzer.

NOTE Agilent 5061-5458 SMA type cables should be used to connect the mixer IF and LO ports to the analyzer. Do not over-tighten the cables. The maximum torque should not exceed 112 N-cm (10 in-lb.)

2. Preset the analyzer. Then set the source as following:

On a Signal Source

• Frequency:	35 GHz
• Amplitude:	0 dBm
• RF Output:	On

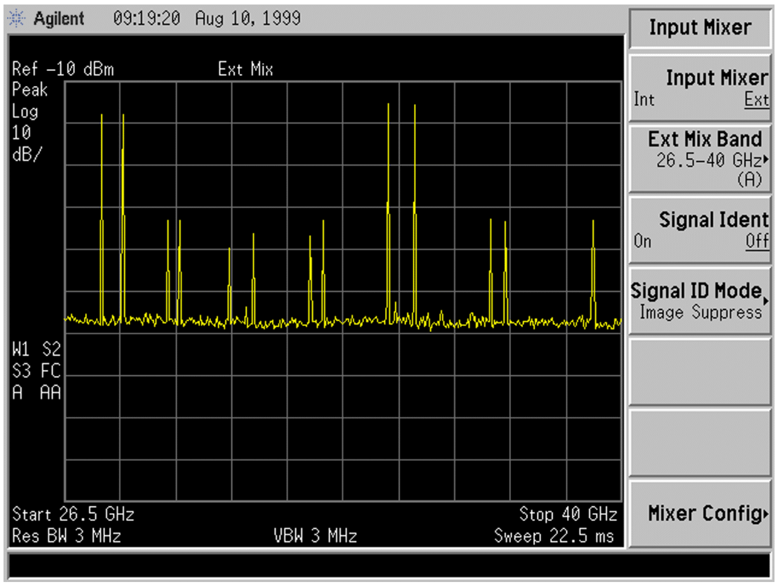
3. Select external mixing.

   Press **Input/Output**, **Input Mixer**, **Input Mixer** (Ext).

The analyzer frequency band will be set to 26.5 – 40 GHz. Multiple mixing products will be shown on the analyzer display. (Other frequency bands could be accessed by pressing **Ext Mix Band**.)

TIP The **Preset** key will take you out of external mixing. If you have changed many settings, pressing **Auto Couple**, **Auto All** returns most settings to their defaults *without* exiting external mixing.

Figure 9-2 Measuring with Unpreselected External Mixer

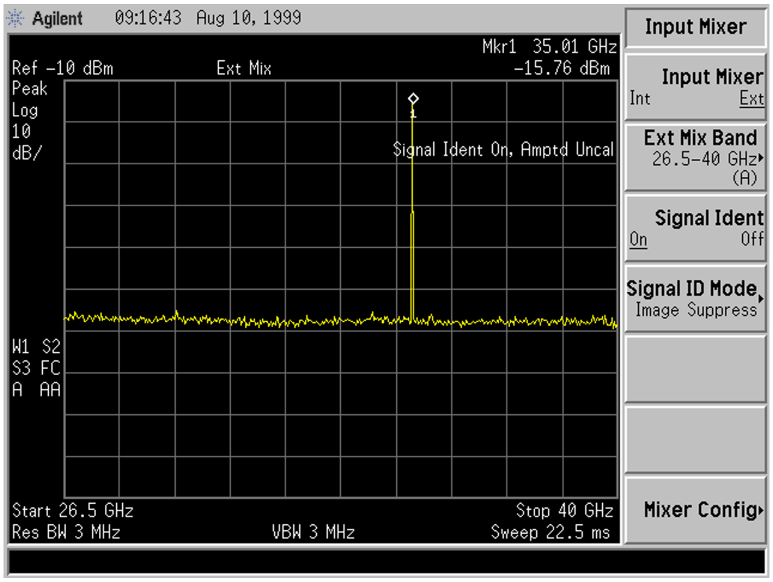


4. Turn on the signal identification feature to identify the valid responses. The default type of signal identification is Image Suppress.

   Press **Input/Output**, **Input Mixer**, **Signal ID** (On).

   Now only the valid response (35 GHz) remains on the display. The signal-identification routine can introduce slight amplitude errors. This is indicated by the message `Signal Ident On, Amptd Uncal`.

Figure 9-3 Signal ID on with Unpreselected External Mixer



5. Put a marker on the valid signal and turn off signal identification before making the final amplitude measurement.

   Press **Peak Search** and **Signal ID** (Off). Then measure the signal amplitude.

Entering Conversion-Loss Correction Data

You may want to correct your measurement for the conversion-loss of the external harmonic mixer that you are using. The amplitude correction feature can be used for this.

1. Press **AMPLITUDE Y Scale**, **More**, **Corrections**.

   You must enter a set of amplitude correction values for the desired frequency range. Select a correction set for use with external mixing. **Other** is the recommended set to use. (Set **Other** to Yes.)

2. Press **Edit** to enter the conversion loss data for the mixer in use. On the Agilent 11970 harmonic mixers, these values are listed on the mixer.

   The data consists of frequency/amplitude pairs. You can enter a single average value for correction over the entire frequency band. Or you can improve frequency response accuracy by entering multiple correction points across the band. Up to 200 points may be defined for each set.

3. Once the desired correction points are entered, you must turn on the correction function. This will improve the display calibration.

   Press **Return**, **Apply Corrections** (Yes) to activate the **Other** corrections.

   Once you have entered the correction set, you should save the set in internal memory or on a floppy disk for future reference.

Setting Mixer Bias

The Agilent 11970 Series harmonic mixers do not require an external bias current. Mixers that require bias can also be used. The conversion loss calibration data for these mixers will be most accurate when the correct bias conditions are applied. Set the bias as follows:

4. To measure a signal, access external mixing and set the band as described in the above procedure.
5. To activate bias press **Input/Output**, **Input Mixer**, **Mixer Config**, **Mixer Bias** (On). A `+I` or `–I` will appear in the display annotation indicating bias is on.
6. Enter the desired bias current in mA.

WARNING **The open-circuit bias voltage can be as great as ±3.5V through a source resistance of 500 ohms. Such voltage levels may appear when recalling an instrument state in which a bias setting has been stored.**

NOTE The bias value that appears on the analyzer display is expressed in terms of short-circuit current (that is, the current that would flow if the `IF INPUT` were shorted to ground). The actual current flowing into the mixer will be less.

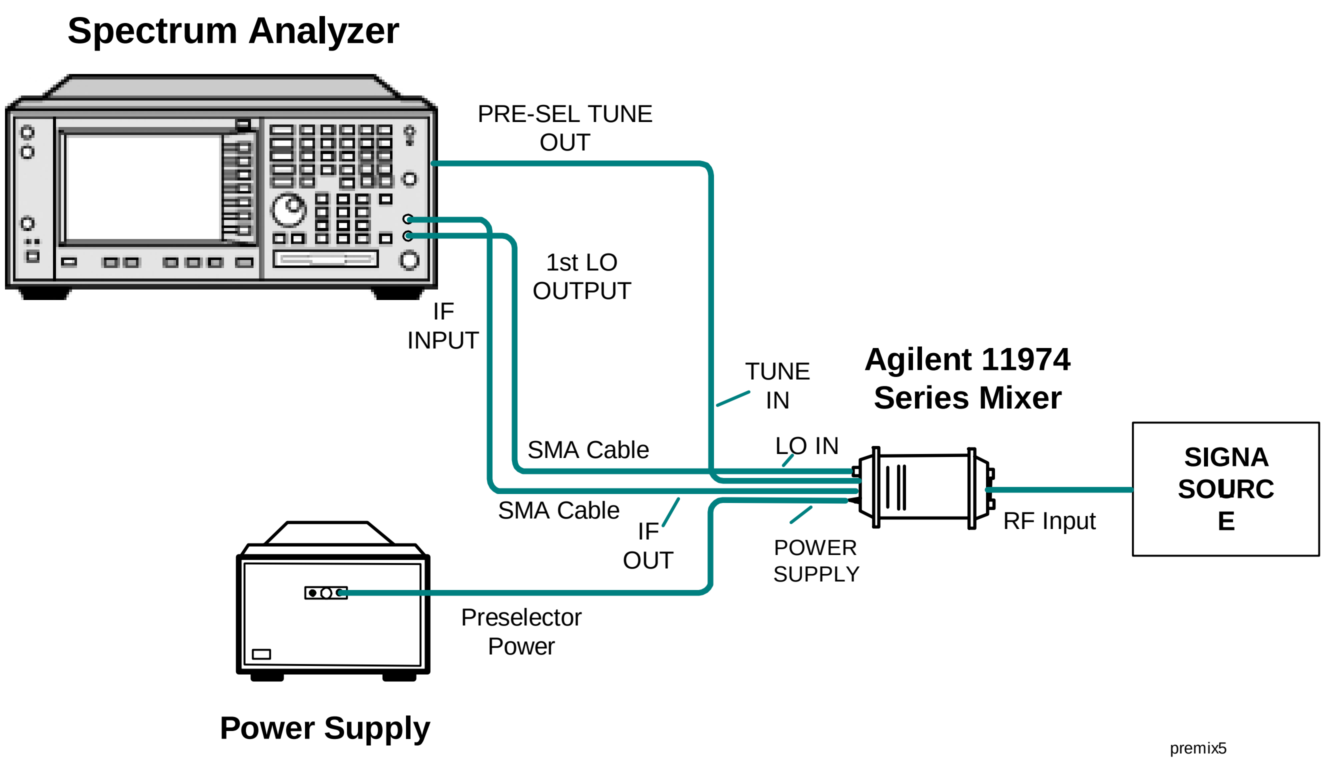
Using Preselected Millimeter-Wave Mixers

For: Agilent 11974 Series Preselected Harmonic Mixers

Before measurements can be made, the preselecting filter inside the harmonic mixer must be aligned so that it tracks the spectrum analyzer frequency tuning.

1. Connect the signal source and preselected mixer to the analyzer as shown.

Figure 9-4 Setup with Preselected External Mixers



Frequency Tracking Alignment

This procedure is used to align the frequency of the preselector filter in the Agilent 11974 to the tuned frequency of the analyzer. This procedure should be followed any time that the Agilent 11974 is connected to a different analyzer. The calibration should be checked periodically.

1. The Agilent 11974 rear-panel switches must be set to scale correctly with the tune signal of the analyzer.

   Set the switches labeled “Agilent 70907B” and “LEDS” to the ON position, and the other two switches to the OFF position.

2. Preset the analyzer. Then configure it for a preselected external mixer.

   Input/Output, **Input Mixer** (Ext), **Mixer Config**, **Mixer Type** (Presel)

3. Set the desired frequency band for your particular mixer.

   Press ‘, **Ext Mix Band**, **A**, **Q**, **U**, or **V**.

4. Set the preselector adjustment to 0 MHz.

   Press **AMPLITUDE**, **Presel Adjust**, 0, **MHz**.

5. Set the analyzer to zero span.

6. Set the analyzer center frequency to the value in Table 9-1, that is for your mixer.

   On the rear panel of the Agilent 11974, adjust the corresponding potentiometer until one or both of the green LEDs are lit.

Table 9-1 Start Frequency Bias Adjustment

Harmonic Mixer	Analyzer Center Frequency	Mixer Potentiometer
11974A	26.5 GHz	“26.5 GHz Adjust”
11974Q	33.0 GHz	“33.0 GHz Adjust”
11974U	40.0 GHz	“40.0 GHz Adjust”
11974V	50.0 GHz	“50.0 GHz Adjust”

7. Change the analyzer center frequency to the value indicated in Table 9-2 and again adjust the corresponding potentiometer on the rear panel of the HP 11974 until one or both of the green LEDs are lit.

Table 9-2 Stop Frequency Bias Adjustment

Harmonic Mixer	Analyzer Center Frequency	Mixer Potentiometer
11974A	40.0 GHz	“40.0 GHz Adjust”
11974Q	50.0 GHz	“50.0 GHz Adjust”
11974U	60.0 GHz	“60.0 GHz Adjust”
11974V	75.0 GHz	“75.0 GHz Adjust”

8. Repeat steps 6 and 7 until the green LEDs are lit at both frequencies without additional adjustments.

Making a Measurement

1. Configure the analyzer for preselected external mixing.

   Press **Input/Output**, **Input Mixer** (Ext), **Mixer Config**, **Mixer Type** (Presel)

2. Use the Frequency Tracking Alignment procedure above to adjust the tracking of the Agilent 11974 to the analyzer.

3. Select the desired mixing band. In this example, we will use an Agilent 11974Q, 33.0 to 50.0 GHz mixer.

   Input/Output, **Input Mixer**, **Ext Mix Band**, **33-50 GHz** (Q)

TIP The **Preset** key will take you out of external mixing. If you have changed many settings, pressing **Auto Couple**, **Auto All** returns most settings to their defaults *without* exiting external mixing.

4. Set the source as following:

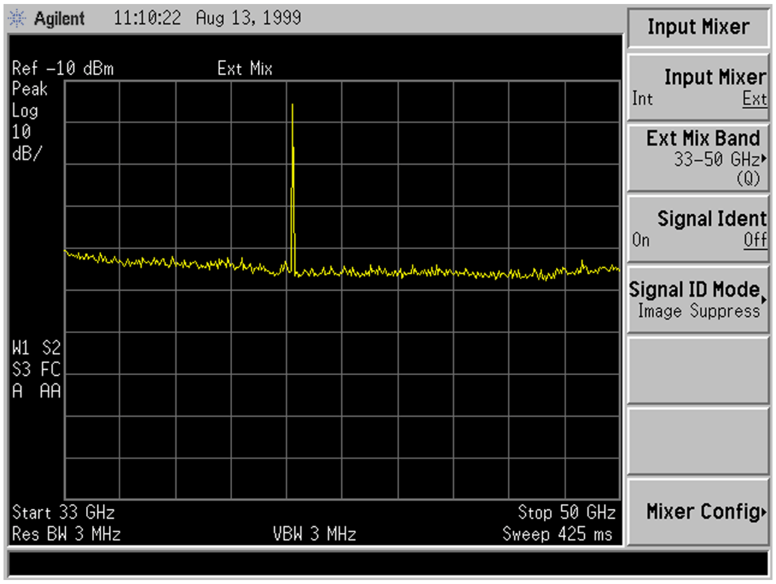
On a Signal Source

•	Frequency:	40 GHz
•	Amplitude:	–15 dBm
•	RF Output:	On

5. Enter the conversion-loss data for the mixer, to calibrate the amplitude of the display. The conversion-loss versus frequency data is on the calibration label on the bottom of the Agilent 11974, or on the supplied calibration sheet.

   Use the procedure "Entering Conversion-Loss Correction Data" on page 81. The full Q-band is shown in Figure 9-5.

Figure 9-5 Q-Band Measurement Display



6. An additional alignment of the preselector must be done at each frequency of interest to optimize the amplitude calibration.

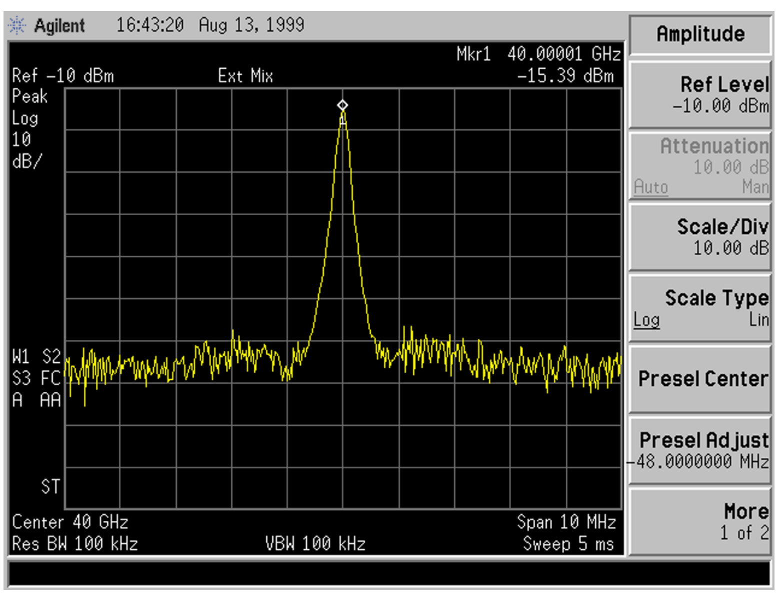
   a. Place a marker on the signal of interest.

   b. Zoom in on the signal by pressing: **SPAN**, **Span Zoom**, 10, **MHz**.

   c. Center the preselector by pressing: **AMPLITUDE**, **Presel Center**.

   The final amplitude measurement can now be read out with the marker. See Figure 9-6.

Figure 9-6



10 Programming Examples

Examples Included:

LabVIEW is a registered trademark of National Instruments Corporation.

Visual Basic is a registered trademark of Microsoft Corporation.

About These Examples

- Many of the examples use the SCPI programming commands, though there are some that use the plug&play or IVI.com drivers.
- Many of the examples are written for an IBM compatible PC.
- There are examples using GPIB and LAN.
- Most of the examples are written in C using the Agilent VISA transition library.

  The VISA transition library must be installed and the GPIB card configured. The Agilent I/O libraries contain the latest VISA transition library and is available at: **`www.agilent.com/iolib`**

Programming Examples

Finding Additional Examples and More Information

These examples are available on the Agilent Technologies PSA Series documentation CD-ROM. They are also available at the URL **http://www.agilent.com/find/psa**

There are additional examples that use the VXI plug&play instrument drivers. These examples are included in the on-line documentation in the driver itself. The driver allows you to use several different programming languages including: VEE, LabVIEW, C, C++, and BASIC. The software driver can be found at the URL **http://www.agilent.com/find/psa**

There are additional examples that use the IntuiLink software. IntuiLink allows you to capture screen and trace data for display and manipulation in the Windows COM environment. These examples are included on the Intuilink CD. The latest version of IntuiLink can also be found at the URL **http://www.agilent.com/find/intuilink**

There is some additional information about the basics of using the C programming language in the C Programming Using VTL section in the Programming Fundamentals chapter of the *User's and Programmer's Reference*.

Using Marker Peak Search

This C programming example (peaksrch.c) can be found on the Documentation CD.

Example:

```
/*********************************************************************/
/* peaksrch.c                                                        */
/* Agilent Technologies 2001                                         */
/*                                  */
/* Using Marker Peak Search and Peak Excursion                       */
/*                                                                   */
/* This example is for the E444xA PSA Spectrum Analyzers             */
/*                                         */
/* This C programming example does the following.                    */
/*                                                                   */
/* - Open a GPIB session at address 18                               */
/* - Select Spectrum Analysis Mode                                   */
/* - Reset & Clear the Analyzer                                      */
/* - Set the analyzer center frequency and span                      */
/* - Set the input port to the 50 MHz amplitude reference            */
/* - Set the analyzer to single sweep mode                           */
/* - Prompt the user for peak excursion level in dBm                 */
/* - Set the peak threshold to user defined level                    */
/* - Trigger a sweep and wait for sweep to complete                  */
/* - Set the marker to the maximum peak                              */
/* - Query and read the marker frequency and amplitude               */
/* - Close the session                                               */
/*********************************************************************/

#include <windows.h>
#include <stdio.h>
#include "visa.h"

ViSession defaultRM, viPSA;
```

Programming Examples

```
ViStatus  errStatus;

void main()
{
    /*Program Variables*/
    ViStatus viStatus  = 0;
    char cEnter = 0;
    int  iResult = 0;
    double dMarkerFreq = 0;
    double dMarkerAmpl = 0;
    float fPeakExcursion =0;
    long lOpc = 0L;

    char *psaSetup =  // PSA setup initialization
        ":INST SA;"                   // Change to Spectrum Analysis mode
             "*RST;*CLS;"      // Reset the device and clear status
        ":SENS:FREQ:CENT 50 MHz;"// Set center freq to 50 MHz
        ":SENS:FREQ:SPAN 50 MHz;"// Set freq span to 50 MHz
        ":SENS:FEED AREF;"// Set input port to internal 50 MHz ref
        ":INIT:CONT 0;"// Set analyzer to single sweep mode
        ":CALC:MARK:PEAK:THR -90";// Set the peak thresold to -90 dBm


    /*Open a GPIB session at address 18.*/
    viStatus=viOpenDefaultRM(&defaultRM);
    viStatus=viOpen(defaultRM,"GPIB0::18",VI_NULL,VI_NULL,&viPSA);
    if(viStatus)
    {
        printf("Could not open a session to GPIB device at address 18!\n");
        exit(0);
    }

    /*Display the program heading */
    printf("\n\t\t Marker Program \n\n" );

    /* Send setup commands to instrument */
```

```
    viPrintf(viPSA,"%s\n",psaSetup);

    /*User enters the peak excursion value */
    printf("\t Enter PEAK EXCURSION level in dBm:  ");
    scanf( "%f",&fPeakExcursion);

    /*Set the peak excursion*/
    viPrintf(viPSA,"CALC:MARK:PEAK:EXC %1fDB \n",fPeakExcursion);

    /*Trigger a sweep and wait for completion*/
    viPrintf(viPSA,"INIT:IMM;*WAI\n");

    /*Set the marker to the maximum peak*/
    viPrintf(viPSA,"CALC:MARK:MAX \n");

    /*Query and read the marker frequency*/
    viQueryf(viPSA,"CALC:MARK:X? \n","%lf",&dMarkerFreq);
    printf("\n\t RESULT: Marker Frequency is: %lf MHz \n\n",dMarkerFreq/10e5);

    /*Query and read the marker amplitude*/
    viQueryf(viPSA,"CALC:MARK:Y?\n","%lf",&dMarkerAmpl);
    printf("\t RESULT: Marker Amplitude is: %lf dBm \n\n",dMarkerAmpl);

    /*Close the session*/
    viClose(viPSA);
    viClose(defaultRM);
}
```

Saving and Recalling Instrument State Data

This C programming example (State.c) can be found on the Documentation CD.

Example:

```
/***************************************************************************
*   State.c
*   Agilent Technologies 2001
*
*   PSA Series Transmitter Tester using VISA for I/O
*   This program shows how to save and recall a state of the instrument
*
*******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "visa.h"

void main ()
{
    /*program variables*/
    ViSession defaultRM, viVSA;
    ViStatus viStatus= 0;

    /*open session to GPIB device at address 18 */
    viStatus=viOpenDefaultRM (&defaultRM);
    viStatus=viOpen (defaultRM, "GPIB0::18::INSTR", VI_NULL,VI_NULL, &viVSA);

    /*check opening session sucess*/
    if(viStatus)
    {
        printf("Could not open a session to GPIB device at address 18!\n");
        exit(0);
    }
```

Programming Examples

```
/*set the instrument to SA mode*/
viPrintf(viVSA, "INST SA\n");

/*reset the instrument */
viPrintf(viVSA, "*RST\n");

/*set the input port to the internal 50Mhz reference source*/
viPrintf(viVSA, "SENS:FEED AREF\n");

/*tune the analyzer to 50MHZ*/
viPrintf(viVSA, "SENS:FREQ:CENT 50E6\n");

/*change the span*/
viPrintf(viVSA, "SENS:FREQ:SPAN 10 MHZ\n");

/*turn the display line on*/
viPrintf(viVSA, "DISP:WIND:TRACE:Y:DLINE:STATE ON\n");

/*change the resolution bandwidth*/
viPrintf(viVSA, "SENS:SPEC:BAND:RES 100E3\n");

/*change the Y Axis Scale/Div*/
viPrintf(viVSA, "DISP:WIND:TRAC:Y:SCAL:PDIV 5\n");

/*Change the display refernece level*/
viPrintf(viVSA, "DISP:WIND:TRAC:Y:SCAL:RLEV -15\n");

/*trigger the instrument*/
viPrintf(viVSA, "INIT:IMM;*WAI\n");

/*save this state in register 10.
!!!Carefull this will overwrite register 10*/

viPrintf(viVSA, "*SAV 10\n");
/*display message*/
```

```
    printf("PSA Programming example showing *SAV,*RCL SCPI commands\n");
    printf("used to save instrument state\n\t\t-----------------------");
    printf("\n\nThe instrument state has been saved to an internal register\n");
    printf("Please observe the display and notice the signal shape\n");
    printf("Then press any key to reset the 
instrument\n\t\t-----------------------");

    /*wait for any key to be pressed*/
    getch();

    /*reset the instrument */
    viPrintf(viVSA, "*RST\n");

    /*set again the input port to the internal 50Mhz reference source*/
    viPrintf(viVSA, "SENS:FEED AREF\n");

    /*display message*/
    printf("\n\nThe instrument was reset to the factory default setting\n");
    printf("Notice the abscence of the signal on the display\n");
    printf("Press any key to recall the saved 
state\n\t\t-----------------------");

    /*wait for any key to be pressed*/
    getch();

    /*recall the state saved in register 10*/
    viPrintf(viVSA, "*RCL 10\n");

    /*display message*/
    printf("\n\nNotice the previous saved instrument settings were restored\n");
    printf("Press any key to terminate the 
program\n\t\t-----------------------\n\n");

    /*wait for any key to be pressed*/
    getch();
```

```
    /*reset the instrument */
    viPrintf(viVSA, "*RST;*wai\n");

    /*Set the instrument to continuous sweep */
    viPrintf(viVSA, "INIT:CONT 1\n");

    /* close session */
    viClose (viVSA);
    viClose (defaultRM);
}
```

Making an ACPR Measurement in cdmaOne

This C programming example (ACPR.c) can be found on the Documentation CD.

Example:

```
/****************************************************************************
*     ACPR.c
*     Adjacent Channel Power Measurement using Power Suite
*     Agilent Technologies 2001
*
*     Instrument Requirements:
*        PSA with firmware version >= A.02.00 or
*        ESA with firmware version >= A.08.00
*
*     Note: You can select which ACPR radio standard you would like by
*           changing the standard for the RADIO:STANDARD command.
*           This example sets the radio standard to IS95.
*
*     Note: For PSA, ensure that you are SA mode before running this program.
*
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "visa.h"

void main ()
{
    /*program variable*/
    ViSession defaultRM, viPSA;
    ViStatus viStatus    = 0;
    ViChar _VI_FAR cResult[2000] = {0};
    int iNum  =0;
    int iSwpPnts = 401;
```


Programming Examples

```
double freq,value;
static ViChar *cToken ;
long lCount=0L;
char sTraceInfo  [1024]= {0};
FILE *fDataFile;
unsigned long lBytesRetrieved;
char *psaSetup =  // PSA setup initialization
    "*RST;*CLS;"  // Reset the device and clear status
    ":INIT:CONT 0;"// Set analyzer to single sweep mode
    ":RADIO:STANDARD IS95";// Set the Radio Standard to IS95

/*open session to GPIB device at address 18 */
viStatus=viOpenDefaultRM (&defaultRM);
viStatus=viOpen (defaultRM, "GPIB0::18::INSTR", VI_NULL,VI_NULL, &viPSA);

/*check opening session sucess*/
if(viStatus)
{
    printf("Could not open a session to GPIB device at address 18!\n");
    exit(0);
}
/*Increase timeout to 20 sec*/
viSetAttribute(viPSA,VI_ATTR_TMO_VALUE,20000);

/*Send setup commands to instrument */
viPrintf(viPSA,"%s\n",psaSetup);

/*Get the center freq from user*/
printf("What is the center carrier frequency in MHz?\n");
scanf( "%lf",&freq);

/*Set the center freq*/
viPrintf(viPSA,"freq:center %lf MHZ\n",freq);

/*Perform an ACPR measurement*/
viQueryf(viPSA,"%s\n", "%#t","READ:ACP?;*wai" , &iNum , cResult);
```

```
    /*Remove the "," from the ASCII data for analyzing data*/
    cToken  = strtok(cResult,",");

    /*Save data to an ASCII to a file, by removing the "," token*/
    fDataFile=fopen("C:\\ACPR.txt","w");
    fprintf(fDataFile,"ACPR.exe Output\nAgilent Technologies 2001\n\n");
    fprintf(fDataFile,"Please read Programer’s Reference for an\n");
    fprintf(fDataFile,"explanation of returned results.\n\n");
    while (cToken != NULL)
        {
            lCount++;
            value = atof(cToken);
            fprintf(fDataFile,"\tReturn value[%d] = %lf\n",lCount,value);
            cToken =strtok(NULL,",");
        }
    fprintf(fDataFile,"\nTotal number of return points of ACPR measurement :[%d]
\n\n",lCount);
    fclose(fDataFile);

    /*print message to the standard output*/
    printf("The The ACPR Measurement Result was saved to C:\\ACPR.txt file\n\n");

    /* Close session */
    viClose (viPSA);
    viClose (defaultRM);
}
```

Programming Examples

Performing Alignments and Getting Pass/Fail Results

This C programming example (SerAlign.c) can be found on the Documentation CD.

Example:

```
/*********************************************************************
*     SerAlign.c
*     Serial Poll Alignment Routine
*     Agilent Technologies 2001
*
*     Instrument Requirements:
*        PSA Series Spectrum Analyzer or
*        ESA Series Spectrum Analyers or
*        VSA Series Transmitter Tester
*
*   This program demonstrates how to
*   1) Perform an instrument alignment.
*   2) Poll the instrument to determine when the operation is complete.
*   3) Query to determine if the alignment was successfuly completed.
*
**********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "visa.h"

void main ()
{
    /*program variables*/
    ViSession defaultRM, viPSA;
    ViStatus viStatus = 0;
    ViUInt16 esr,stat;
    long lResult = 0;
```

```
long lOpc = 0;
char cEnter = 0;

/*open session to GPIB device at address 18 */
viStatus=viOpenDefaultRM (&defaultRM);
viStatus=viOpen (defaultRM, "GPIB0::18::INSTR", VI_NULL,VI_NULL,&viPSA);

/*check opening session sucess*/
if(viStatus)
{
    printf("Could not open a session to GPIB device at address 18!\n");
    exit(0);
}

/*increase timeout to 60 sec*/
viSetAttribute(viPSA,VI_ATTR_TMO_VALUE,60000);

/*Clear the analyzer*/
viClear(viPSA);

/*Clear all event registers*/
viPrintf(viPSA, "*CLS\n");

/* Set the Status Event Enable Register */
viPrintf(viPSA, "*ESE 1\n");

/*Initiate self-alignment*/
viPrintf(viPSA, "CAL:ALL\n");

/* Send the Operation complete command so that the
   stand event register will be set to 1 once
   the pending alignment command is complete */
viPrintf(viPSA, "*OPC\n");

/* print message to standard output */
printf("Performing self-alignment.\n");
```

```
    /* Serial pole the instrument for operation complete */
    while(1)
    {
        viQueryf(viPSA,"*ESR?\n","%ld",&esr);
        printf(".");
        if (esr & 1) break;//look for operation complete bit
        Sleep (1000);// wait 1000ms before polling again
    }

    /* Query the Status Questionable Condition Register */
    viQueryf(viPSA,":STAT:QUES:CAL:COND?\n","%ld",&stat);

    /*Determine if alignment was successful*/
    if (stat)
        printf("\nAlignment not successful\n\n");
    else
        printf("\nAlignment successful\n\n");

    /*reset timeout to 5 sec*/
    viSetAttribute(viPSA,VI_ATTR_TMO_VALUE,5000);

    /*print message to the standard output*/
    printf("Press Return to exit program \n\n");
    scanf("%c",&cEnter);

    /* Close session */
    viClose (viPSA);
    viClose (defaultRM);
}
```

Saving Binary Trace Data

This C programming example (Trace.c) can be found on the Documentation CD.

This example uses Option B7J.

Example:

```
/***************************************************************************
*     Trace.c
*     Agilent Technologies 2001
*
*     Instrument Requirements:
*        E444xA with option B7J and firmware version >= A.02.00 or
*        E4406A with firmware version >= A.05.00
*
*     This Program shows how to get and save binary trace data in Basic mode
*     The results are saved to C:\trace.txt
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "visa.h"

void main ()
{
    /*program variable*/
    ViSession defaultRM, viPSA;
    ViStatus viStatus= 0;
    char sBuffer[80]= {0};
    char dummyvar;
    FILE *fTraceFile;
    long lNumberPoints= 0;
    long lNumberBytes= 0;
    long lLength= 0;
    long i = 0;
```

```
long lOpc  = 0L;
unsigned long lBytesRetrieved;
ViReal64 adTraceArray[10240];

char *psaSetup =/* setup commands for VSA/PSA */
    ":INST BASIC;"/* Set the instrument mode to Basic */
    "*RST;*CLS;"/* Reset the device and clear status */
    ":INIT:CONT 0;"/* Set analyzer to single measurement mode */
    ":FEED AREF;"/* set the input port to the internal
               50MHz reference source */
    ":DISP:FORM:ZOOM1;"/* zoom the spectrum display */
    ":FREQ:CENT 50E6;"/* tune the analyzer to 50MHz */
    ":FORM REAL,64;"/* Set the ouput format to a binary format */
    ":FORM:BORD SWAP;"/* set the binary byte order to SWAP (for PC) */
    ":INIT:IMM;";/* trigger a spectrum measurement */

/*open session to GPIB device at address 18 */
viStatus=viOpenDefaultRM (&defaultRM);
viStatus=viOpen (defaultRM, "GPIB0::18::INSTR", VI_NULL,VI_NULL, &viPSA);

/*check opening session sucess*/
if(viStatus)
{
    printf("Could not open a session to GPIB device at address 18!\n");
    exit(0);
}

/* Set I/O timeout to ten seconds */
viSetAttribute(viPSA,VI_ATTR_TMO_VALUE,10000);

/* Send setup commands to instrument */
viPrintf(viPSA,"%s\n",vsaSetup);

/* Query the instrument for Operation complete */
viQueryf(viPSA,"*OPC?\n", "%d", &lOpc);
```

Programming Examples

```
/* fetch the spectrum trace data*/
viPrintf(viPSA,"FETC:SPEC7?\n");

/*print message to the standard output*/
printf("Getting the spectrum trace in binary format...\nPlease wait...\n\n");

/* get number of bytes in length of postceeding trace data
   and put this in sBuffer*/
viRead (viPSA,(ViBuf)sBuffer,2,&lBytesRetrieved);

/* Put the trace data into sBuffer */
viRead (viPSA,(ViBuf)sBuffer,sBuffer[1] - '0',&lBytesRetrieved);

/* append a null to sBuffer */
sBuffer[lBytesRetrieved] = 0;

/* convert sBuffer from ASCII to integer */
lNumberBytes = atoi(sBuffer);

/* calculate the number of points given the number of byte in the trace
   REAL 64 binary format means each number is represented by 8 bytes*/
lNumberPoints = lNumberBytes/sizeof(ViReal64);

/*get and save trace in data array */
viRead (viPSA,(ViBuf)adTraceArray,lNumberBytes,&lBytesRetrieved);

/* read the terminator character and discard */
viRead (viPSA,(ViBuf)sBuffer,1, &lLength);

/*print message to the standard output*/
printf("Querying instrument to see if any errors in Queue.\n");

/* loop until all errors read */
do
{
    viPrintf (viPSA,"SYST:ERR?\n");/* check for errors */
```

Programming Examples

```
        viRead (viPSA,(ViBuf)sBuffer,80,&lLength);/* read back last error message
*/
        sBuffer[lLength] = 0;                    /* append a null to byte count */
        printf("%s\n",sBuffer);                 /* print error buffer to display */
    } while (sBuffer[1] != '0');

    /* set the analyzer to continuous mode for manual use */
    viPrintf(viPSA, "INIT:CONT 1\n");

    /*save trace data to an ASCII file*/
    fTraceFile=fopen("C:\\Trace.txt","w");
    fprintf(fTraceFile,"Trace.exe Output\nAgilent Technologies 2001\n\n");
    fprintf(fTraceFile,"List of %d points of the averaged spectrum
trace:\n\n",lNumberPoints);
    for (i=0;i<lNumberPoints;i++)
       fprintf(fTraceFile,"\tAmplitude of point[%d] = %.2lf
dBm\n",i+1,adTraceArray[i]);
    fclose(fTraceFile);

    /*print message to the standard output*/
    printf("The %d trace points were saved to C:\\Trace.txt
file\n\n",lNumberPoints);

    /* Send message to standard output */
    printf("\nPress Enter to set analyzer's input port back to RF.\n");
    scanf("%c",&dummyvar);

    /* set the input port to RF */
    viPrintf(viPSA, "feed rf\n");

    /* Close session */
    viClose (viPSA);
    viClose (defaultRM);
}
```

Programming Examples

Making a Power Calibration for a GSM Mobile Handset

This C programming example (powercal.c) can be found on the Documentation CD.

This program uses Basic mode which is optional -B7J- in the PSA Series spectrum analyzers and is standard in the E4406A Vector Signal Analyzer. It uses the Waveform measurement with the `CALC:DATA2:COMP? DME` command to return the power of 75 consecutive GSM/EDGE bursts. The DME (dB Mean) parameter returns the average of the dB trace values. The DME parameter is only available in later version of instrument firmware ≥ A.05.00 for PSA and ≥ A.07.00 for VSA. Earlier instruments see the “Using the CALCulate:DATA:COMPress? RMS Command” example.

This program also demonstrates how to serial poll the "Waiting for Trigger" status bit to determine when to initiate the GSM phone. The data results are placed in an ASCII file (powercal.txt).

The program can also be adapted to perform W-CDMA Downlink Power Control measurements in the code domain power `Symbol Power` view. In essence, you can average any stepped power measurement trace using this method.

Example:

```
/*********************************************************************************
*     powercal.c
*     Agilent Technologies 2003
*
*     This program demonstrates the process of using the Waveform
*     measurement and the CALC:DATA2:COMP? DME command to return the power
*     of 75 consecutive GSM/EDGE bursts.
*     The DME (db Mean) parameter returns the average of the dB trace values.
*
*     This program also demonstrates how to serial poll the "Waiting
*     for Trigger" Status bit to determine when to initiate the GSM phone
*     The data results are placed in an ASCII file, powercal.txt
*
*     This program can also be adapted to perform W-CDMA Downlink PowerControl
*     measurements in the Code Domain Power Symbol Power View.  In essence,
*     you can average any stepped power measurement trace using this method.
```

Programming Examples

```
*
*     Instrument Requirements:
*        E444xA with option B7J and firmware version >= A.05.00 or
*        E4406A with firmware version >= A.07.00 or
*
*     Signal Source Setup:
*        Set up GSM/EDGE frame for either 1, 2, 4, or eight slots per frame.
*        When configuring two slots per frame, turn on slots 1 and 5
*        When configuring four slots per frame, turn on slots 1,3,5, and 7.
*        Set frame repeat to Single.
*        Set the signal amplitude to -5 dBm.
*        Set the signal source frequency to 935.2 MHz
*
*     CALC:DATA2:COMP? DME parameters:
*        soffset = 25us (This avoids averaging data points when the burst
*                        is transitioning on.)
*        length = 526us (Period over which the burst is averaged)
*        roffset = 4.6153846 ms / slots per frame (Repitition interval of burst)
*********************************************************************************
****/
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include "c:\program files\visa\winnt\include\visa.h"

void main ()
{
   /*program variable*/
   ViSession defaultRM, viVSA;
   ViStatus viStatus= 0;
   ViUInt16 stb;
   FILE *fDataFile;
   long lthrowaway,lbursts;
   long lNumberPoints= 0;
   long lNumberBytes= 0;
```


Programming Examples

```
long lLength     = 0;
long i              = 0;
long lOpc  = 0L;
double sweeptime    = 0;
double burstinterval= 0;
unsigned long lBytesRetrieved;
ViReal64 adDataArray[100];
char sBuffer[80]= {0};

char *basicSetup = // measurement setup commands for VSA/PSA
    ":INST:SEL BASIC;"// Put the instrument in Basic Mode
     "*RST;"// Preset the instrument
    "*CLS;     "//Clear the status byte
    ":STAT:OPER:ENAB 32;"       //Enable Status Operation
    ":DISP:ENAB 0;"// Turn the Display off (improves Speed)
    ":FORM REAL,64;"// Set the ouput format to binary
    ":FORM:BORD SWAP;"// set the binary byte order to SWAP (for PC)
    ":CONF:WAV;"// Changes measurement to Waveform
    ":INIT:CONT 0;"// Puts instrument in single measurement mode
    ":CAL:AUTO OFF;"//Turn auto align off
    ":FREQ:CENTER 935.2MHz;"//Set Center Freq to 935.2MHz
    ":WAV:ACQ:PACK MED;"//Set DataPacking to Medium
    ":WAV:BAND:TYPE FLAT;"//Select Flattop RBW Filter
    ":WAV:DEC:FACT 4;"//Set Decimation Factor to 4
    ":WAV:DEC:STAT ON;"//Turn Decimation On
    ":DISP:WAV:WIND1:TRAC:Y:RLEV 5;" //Set referance level to 5 dBm
    ":WAV:BWID:RES 300kHz;"//Set Res bandwith filter to 300kHz
    ":POW:RF:ATT 5;"//Set 5dB of internal attenuation
    ":WAV:ADC:RANG P0;"//Set ADC Range to P0, This is
                      //necessary to prevent autoranging
    ":WAV:TRIG:SOUR IF;"//Set Trigger source to IF burst
    ":TRIG:SEQ:IF:LEV -20;";//Set IF Trig level to -20dB

/*open session to GPIB device at address 18 */
viStatus=viOpenDefaultRM (&defaultRM);
viStatus=viOpen (defaultRM, "GPIB0::18", VI_NULL,VI_NULL,&viVSA);
```

Programming Examples

```
/*check opening session sucess*/
if(viStatus)
{
    printf("Could not open a session to GPIB device at address 18!\n");
    exit(0);
}

/* Set I/O timeout to ten seconds */
viSetAttribute(viVSA,VI_ATTR_TMO_VALUE,10000);

viClear(viVSA);//send device clear to instrument

/*print message to the standard output*/
printf("Enter number of bursts per frame (1,2,4 or 8): ");
scanf( "%ld",&lbursts);

/* Send setup commands to instrument */
viPrintf(viVSA,"%s\n",basicSetup);

/* Calculate sweep time and set it*/
burstinterval = 4.6153846 / 1000.00 / lbursts;
sweeptime= burstinterval * 75.0;

viPrintf(viVSA,":WAV:SWE:TIME %fs\n",sweeptime);

/* Clear status event register */
viQueryf(viVSA,"STAT:OPER:EVENT?\n","%ld",&lthrowaway);

/* Initiate the waveform measurement and get the instrument ready
   to calculate the mean RMS I/Q voltage in each burst
   (We will convert these discreate values into Mean dBm Power values) */

viPrintf(viVSA,"INIT:IMM\n");

/* Serial poll the instrument to determine when it is waiting for
```

Programming Examples

```
   trigger and the GSM phone can be told to send its 75 bursts. */

while(1)
{
    viReadSTB(viVSA,&stb);  //read status byte
    if (stb & 128) break;   //look for "waiting for trigger" bit
    printf("Waiting on Analyzer...\n");
    Sleep (50); // wait 50 ms between each serial poll
}
/*print message to the standard output*/
printf("Analyzer is Ready!\n\nWaiting for phone to trigger...\n\n");

/*Querry for Operation Complete */
viQueryf(viVSA,"*OPC?\n", "%d", &lOpc);

 /*Use the CALC:DATA0:COMP command to return the average power in each burst*/
viPrintf(viVSA,":CALC:DATA2:COMP? DME,25E-6,526E-6,%f\n",burstinterval);

/* get number of bytes in length of postceeding data and put this in sBuffer*/
viRead (viVSA,(ViBuf)sBuffer,2,&lBytesRetrieved);
printf("Getting the burst data in binary format...\n\n");

/* Put the returned data into sBuffer */
viRead (viVSA,(ViBuf)sBuffer,sBuffer[1] - '0',&lBytesRetrieved);

/* append a null to sBuffer */
sBuffer[lBytesRetrieved] = 0;

/* convert sBuffer from ASCII to integer */
lNumberBytes = atoi(sBuffer);

/*calculate the number of returned values given the number of bytes.
  REAL 64 binary data means each number is represented by 8 bytes */
lNumberPoints = lNumberBytes/sizeof(ViReal64);

/*get and save returned data into an array */
```

Programming Examples

```
    viRead (viVSA,(ViBuf)adDataArray,lNumberBytes,&lBytesRetrieved);

    /* read the terminator character and discard */
    viRead (viVSA,(ViBuf)sBuffer,1, &lthrowaway);

    /*print message to the standard output*/
    printf("Querying instrument to see if any errors in Queue.\n");

    /* loop until all errors read */
    do
    {
       viPrintf (viVSA,"SYST:ERR?\n");           /* check for errors */
       viRead (viVSA,(ViBuf)sBuffer,80,&lLength);/* read back last error message */
       sBuffer[lLength] = 0;                     /* append a null to byte count */
       printf("%s\n",sBuffer);                  /* print error buffer to display */
    } while (sBuffer[1] != '0');

    /* Turn the Display of the instrument back on */
    viPrintf(viVSA,"DISP:ENAB 1\n");

    /*save result data to an ASCII file*/
    fDataFile=fopen("powercal.txt","w");
    fprintf(fDataFile,"powercal.exe Output\nAgilent Technologies 2003\n\n");
    fprintf(fDataFile,"Power of %d GSM/EDGE bursts:\n",lNumberPoints);
    fprintf(fDataFile,"(%d burst(s) per frame):\n\n",lbursts);
    for (i=0;i<lNumberPoints;i++)
    {
        fprintf(fDataFile,"\tPower of burst[%d] = %.2lf dBm\n",i+1,adDataArray[i]);
    }
    fclose(fDataFile);
    /*print message to the standard output*/
    printf("The %d burst powers were saved to powercal.txt
file.\n\n",lNumberPoints);

    viClose (viVSA);
    viClose (defaultRM);
```

Programming Examples

```
}
```

Using the CALCulate:DATA:COMPress? RMS Command

This C programming example (calcomp.c) can be found on the Documentation CD.

This program uses the `CALCulate:DATA:COMPress? RMS` command to average the voltage trace data to calculate power of consecutive GSM bursts. Older instrument firmware does *not* support the newer DME parameter described in the previous example. You will have to use the technique in this example to calculate the dB mean. This example uses the Waveform measurement in the Basic mode. Basic mode is optional -B7J- in the PSA Series spectrum analyzers and is standard in the E4406A Vector Signal Analyzer.

The `CALC:DATA2:COMP? RMS` command is used to return the power of 1 to 150 consecutive GSM/EDGE bursts. The RMS parameter returns the average of the voltage trace values. These measured values are then converted to dBm values.

This program also demonstrates how to serial poll Serial poll the instrument to determine when the Message Available status bit is set.

Example:

```
/***************************************************************************
calcomp.c
*     Agilent Technologies 2001
*
*     This program demonstrates the process of using the Waveform
*     measurement and the CALC:DATA0:COMP? RMS command to return the power
*     of 1 to 450 consecutive GSM/EDGE bursts (one burst per frame).
*     The data results are placed in an ASCII file, C:\calccomp.txt
*
*     Instrument Requirements:
*        E444xA with option B7J and firmware version >= A.02.00 or
*        E4406A with firmware version >= A.05.00 or
*
*     Signal Source Setup:
*        Turn on 1 slot per GSM/EDGE frame.
*        Set frame repeat to Continous.
*        Set the signal amplitude to -5 dBm.
```

Programming Examples

```
*        Set the signal source frequency to 935.2 MHz
*
*     CALC:DATA0:COMP? RMS parameters:
*        soffset = 25us (This avoids averaging data points when the burst
*                        is transitioning on.)
*        length = 526us (Period over which the power of the burst is averaged)
*        roffset = 4.165 ms (Repition interval of burst.  For this example
*                            it is equal to one GSM frame: 4.165 ms.)
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include "visa.h"

void main ()
{
    /*program variable*/
    ViSession defaultRM, viPSA;
    ViStatus viStatus= 0;
    ViUInt16 stb;
    FILE *fDataFile;
    long lthrowaway,lbursts;
    long lNumberPoints= 0;
    long lNumberBytes= 0;
    long lLength     = 0;
    long i              = 0;
    long lOpc  = 0L;
    double sweeptime    = 0;
    unsigned long lBytesRetrieved;
    ViReal64 adDataArray[500];
    ViReal64 adPowerArray[500];
    char sBuffer[80]= {0};

    char *basicSetup = // measurement setup commands for VSA/PSA
        ":INST:SEL BASIC;"// Put the instrument in Basic Mode
```


Programming Examples

```
     "*RST;"// Preset the instrument
    "*CLS;"    //Clear the status byte
    ":DISP:ENAB 0;"// Turn the Display off (improves Speed)
    ":FORM REAL,64;"// Set the ouput format to binary
    ":FORM:BORD SWAP;"// set the binary byte order to SWAP (for PC)
    ":CONF:WAV;"// Changes measurement to Waveform
    ":INIT:CONT 0;"// Puts instrument in single measurement mode
    ":CAL:AUTO OFF;"//Turn auto align off
    ":FREQ:CENTER 935.2MHz;"//Set Center Freq to 935.2MHz
    ":WAV:ACQ:PACK MED;"//Set DataPacking to Medium
    ":WAV:BAND:TYPE FLAT;"//Select Flattop RBW Filter
    ":WAV:DEC:FACT 4;"//Set Decimation Factor to 4
    ":WAV:DEC:STAT ON;"//Turn Decimation On
    ":DISP:WAV:WIND1:TRAC:Y:RLEV 5;" //Set referance level to 5 dBm
    ":WAV:BWID:RES 300kHz;"//Set Res bandwith filter to 300kHz
    ":POW:RF:ATT 5;"//Set 5dB of internal attenuation
    ":WAV:TRIG:SOUR IF;"//Set Trigger source to IF burst
    ":TRIG:SEQ:IF:LEV -20;";//Set IF Trig level to -20dB

/*open session to GPIB device at address 18 */
viStatus=viOpenDefaultRM (&defaultRM);
viStatus=viOpen (defaultRM, "GPIB0::18", VI_NULL,VI_NULL,&viPSA);

/*check opening session sucess*/
if(viStatus)
{
    printf("Could not open a session to GPIB device at address 18!\n");
    exit(0);
}

/* Set I/O timeout to ten seconds */
viSetAttribute(viPSA,VI_ATTR_TMO_VALUE,10000);

viClear(viPSA);//send device clear to instrument

/*print message to the standard output*/
```

Programming Examples

```
printf("Enter number of bursts (1 to 450) to calculate mean power for:  ");
scanf( "%ld",&lbursts);

/* Send setup commands to instrument */
viPrintf(viPSA,"%s\n",basicSetup);

/* Calculate sweep time and set it*/
sweeptime=4.6153846*lbursts;
viPrintf(viPSA,":WAV:SWE:TIME %fms\n",sweeptime);

/* Clear status event register */
viQueryf(viPSA,"STAT:OPER:EVENT?\n","%ld",&lthrowaway);

/* Initiate the waveform measurement */
viPrintf(viPSA,"INIT:IMM\n");

/* Query the instrument for Operation complete */
viQueryf(viPSA,"*OPC?\n", "%d", &lOpc);

/* Have the instrument calculate the mean RMS I/Q voltage in each burst
   (We will convert these discreate values into Mean dBm Power values) */
viPrintf (viPSA, ":CALC:DATA0:COMP? rms,25E-6,526E-6,4.61538461538E-3\n");

/* Serial poll the instrument to determine when Message Available
   Status Bit is set.  The instrument’s output buffer will then
   contain the measurement results*/
i=0;
while(1)
{
    i++;
    viReadSTB(viPSA,&stb);  //read status byte
    if (stb & 16) break;    //look for message available bit
    Sleep (20);  // wait 100ms before polling again
}

/*print message to the standard output*/
```

Programming Examples

```
printf("\nMessage Available statuts bit set after %ld serial poles.\n\n",i);
printf("Getting the burst data in binary format...\nPlease wait...\n\n");

/* get number of bytes in length of postceeding data
   and put this in sBuffer*/
viRead (viPSA,(ViBuf)sBuffer,2,&lBytesRetrieved);

/* Put the returned data into sBuffer */
viRead (viPSA,(ViBuf)sBuffer,sBuffer[1] - '0',&lBytesRetrieved);

/* append a null to sBuffer */
sBuffer[lBytesRetrieved] = 0;

/* convert sBuffer from ASCII to integer */
lNumberBytes = atoi(sBuffer);

/*calculate the number of returned values given the number of bytes.
  REAL 64 binary data means each number is represented by 8 bytes */
lNumberPoints = lNumberBytes/sizeof(ViReal64);

/*get and save returned data into an array */
viRead (viPSA,(ViBuf)adDataArray,lNumberBytes,&lBytesRetrieved);

/* read the terminator character and discard */
viRead (viPSA,(ViBuf)sBuffer,1, &lthrowaway);

/*print message to the standard output*/
printf("Querying instrument to see if any errors in Queue.\n");

/* loop until all errors read */
do
{
   viPrintf (viPSA,"SYST:ERR?\n");           /* check for errors */
   viRead (viPSA,(ViBuf)sBuffer,80,&lLength);/* read back last error message */
   sBuffer[lLength] = 0;                     /* append a null to byte count */
   printf("%s\n",sBuffer);                  /* print error buffer to display */
```


Programming Examples

```
    } while (sBuffer[1] != '0');


    /* Turn the Display of the instrument back on */
    viPrintf(viPSA,"DISP:ENAB 1\n");


    /*save result data to an ASCII file*/
    fDataFile=fopen("C:\\calccomp.txt","w");
    fprintf(fDataFile,"Calccomp.exe Output\nAgilent Technologies 2001\n\n");
    fprintf(fDataFile,"Power of %d GSM/EDGE bursts:\n\n",lNumberPoints);
    for (i=0;i<lNumberPoints;i++)
    {
        /* Convert RMS voltage for each burst to Mean Power in dBm */
        adPowerArray[i]=10*log10(10*adDataArray[i]*adDataArray[i]);
        fprintf(fDataFile,"\tPower of burst[%d] = %.2lf
dBm\n",i+1,adPowerArray[i]);
    }
    fclose(fDataFile);
    /*print message to the standard output*/
    printf("The %d burst powers were saved to C:\\calccomp.txt
file.\n\n",lNumberPoints);


    viClose (viPSA);
    viClose (defaultRM);
}
```

Programming Examples

Programming Examples

Using C Over Socket LAN (UNIX)

This C programming example (socketio.c) compiles in the HP-UX UNIX environment. It is portable to other UNIX environments with only minor changes.

In UNIX, LAN communication via sockets is very similar to reading or writing a file. The only difference is the openSocket( ) routine, which uses a few network library routines to create the TCP/IP network connection. Once this connection is created, the standard fread( ) and fwrite( ) routines are used for network communication.

In Windows, the routines send( ) and recv( ) must be used, since fread( ) and fwrite( ) may not work on sockets.

The program reads the analyzer's host name from the command line, followed by the SCPI command. It then opens a socket to the analyzer, using port 5025, and sends the command. If the command appears to be a query, the program queries the analyzer for a response, and prints the response.

This example program can also be used as a utility to talk to your analyzer from the command prompt on your UNIX workstation or Windows 95 PC, or from within a script.

This program is also available on your documentation CD ROM.

Example:

```
/***************************************************************************
*  $Header: socketio.c,v 1.5 96/10/04 20:29:32 roger Exp $
*  $Revision: 1.5 $
*  $Date: 96/10/04 20:29:32 $
*
*  $Contributor:     LSID, MID $
*
*  $Description:     Functions to talk to an Agilent E4440A spectrum
*                    analyzer via TCP/IP.  Uses command-line arguments.
*
*                    A TCP/IP connection to port 5025 is established and
*                    the resultant file descriptor is used to "talk" to the
*                    instrument using regular socket I/O mechanisms. $
*
*
```

```
*
*  E4440A Examples:
*
*   Query the center frequency:
*        lanio 15.4.43.5 ’sens:freq:cent?’
*
* Query X and Y values of marker 1 and marker 2 (assumes they are on):
*        lanio myinst ’calc:spec:mark1:x?;y?; :calc:spec:mark2:x?;y?’
*
*   Check for errors (gets one error):
*        lanio myinst ’syst:err?’
*
*   Send a list of commands from a file, and number them:
*        cat scpi_cmds | lanio -n myinst
*
****************************************************************************
*
*  This program compiles and runs under
*     - HP-UX 10.20 (UNIX), using HP cc or gcc:
*           + cc -Aa    -O -o lanio  lanio.c
*           + gcc -Wall -O -o lanio  lanio.c
*
*     - Windows 95, using Microsoft Visual C++ 4.0 Standard Edition
*     - Windows NT 3.51, using Microsoft Visual C++ 4.0
*           + Be sure to add  WSOCK32.LIB  to your list of libraries!
*           + Compile both lanio.c and getopt.c
*           + Consider re-naming the files to lanio.cpp and getopt.cpp
*
*  Considerations:
*     - On UNIX systems, file I/O can be used on network sockets.
*       This makes programming very convenient, since routines like
*       getc(), fgets(), fscanf() and fprintf() can be used.  These
*       routines typically use the lower level read() and write() calls.
*
*     - In the Windows environment, file operations such as read(), write(),
*       and close() cannot be assumed to work correctly when applied to
```

Programming Examples

```
 *       sockets.  Instead, the functions send() and recv() MUST be used.
 */

/* Support both Win32 and HP-UX UNIX environment */
#ifdef _WIN32     /* Visual C++ 4.0 will define this */
#  define WINSOCK
#endif

#ifndef WINSOCK
#  ifndef _HPUX_SOURCE
#  define _HPUX_SOURCE
#  endif
#endif

#include <stdio.h>         /* for fprintf and NULL  */
#include <string.h>        /* for memcpy and memset */
#include <stdlib.h>        /* for malloc(), atol() */
#include <errno.h>         /* for strerror          */

#ifdef WINSOCK

#include <windows.h>

#  ifndef _WINSOCKAPI_
#  include <winsock.h>   // BSD-style socket functions
#  endif

#else /* UNIX with BSD sockets */

#  include <sys/socket.h>    /* for connect and socket*/
#  include <netinet/in.h>    /* for sockaddr_in       */
#  include <netdb.h>         /* for gethostbyname     */

#  define SOCKET_ERROR (-1)
#  define INVALID_SOCKET (-1)
```


Programming Examples

```
   typedef  int SOCKET;

#endif /* WINSOCK */

#ifdef WINSOCK
  /* Declared in getopt.c.  See example programs disk. */
  extern char *optarg;
  extern int  optind;
  extern int getopt(int argc, char * const argv[], const char* optstring);
#else
#  include <unistd.h>           /* for getopt(3C) */
#endif

#define COMMAND_ERROR  (1)
#define NO_CMD_ERROR  (0)

#define SCPI_PORT  5025
#define INPUT_BUF_SIZE (64*1024)


/***************************************************************************
 * Display usage
 ***************************************************************************/
static void usage(char *basename)
{
    fprintf(stderr,"Usage: %s [-nqu] <hostname> [<command>]\n", basename);
    fprintf(stderr,"       %s [-nqu] <hostname> < stdin\n", basename);
    fprintf(stderr,"  -n, number output lines\n");
    fprintf(stderr,"  -q, quiet; do NOT echo lines\n");
    fprintf(stderr,"  -e, show messages in error queue when done\n");
}



#ifdef WINSOCK
int init_winsock(void)
```


Programming Examples

```
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD(1, 1);
    wVersionRequested = MAKEWORD(2, 0);

    err = WSAStartup(wVersionRequested, &wsaData);

    if (err != 0) {
        /* Tell the user that we couldn't find a useable */
        /* winsock.dll.     */
        fprintf(stderr, "Cannot initialize Winsock 1.1.\n");
        return -1;
    }
    return 0;
}

int close_winsock(void)
{
    WSACleanup();
    return 0;
}
#endif /* WINSOCK */



/***************************************************************************
 *
 > $Function: openSocket$
 *
 * $Description:  open a TCP/IP socket connection to the instrument $
 *
 * $Parameters:  $
 *    (const char *) hostname . . . . Network name of instrument.
 *                                    This can be in dotted decimal notation.
```

```
 *    (int) portNumber  . . . . . . . The TCP/IP port to talk to.
 *                                    Use 5025 for the SCPI port.
 *
 * $Return:     (int)  . . . . . . . . A file descriptor similar to open(1).$
 *
 * $Errors:     returns -1 if anything goes wrong $
 *
 ***************************************************************************/
SOCKET openSocket(const char *hostname, int portNumber)
{
    struct hostent *hostPtr;
    struct sockaddr_in peeraddr_in;
    SOCKET s;


    memset(&peeraddr_in, 0, sizeof(struct sockaddr_in));

    /***********************************************/
    /* map the desired host name to internal form. */
    /***********************************************/
    hostPtr = gethostbyname(hostname);
    if (hostPtr == NULL)
    {
        fprintf(stderr,"unable to resolve hostname '%s'\n", hostname);
        return INVALID_SOCKET;
    }

    /*******************/
    /* create a socket */
    /*******************/
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == INVALID_SOCKET)
    {
        fprintf(stderr,"unable to create socket to '%s': %s\n",
                hostname, strerror(errno));
        return INVALID_SOCKET;
```

```
    }

    memcpy(&peeraddr_in.sin_addr.s_addr, hostPtr->h_addr, hostPtr->h_length);
    peeraddr_in.sin_family = AF_INET;
    peeraddr_in.sin_port = htons((unsigned short)portNumber);

    if (connect(s, (const struct sockaddr*)&peeraddr_in,
                sizeof(struct sockaddr_in)) == SOCKET_ERROR)
    {
        fprintf(stderr,"unable to create socket to '%s': %s\n",
                hostname, strerror(errno));
        return INVALID_SOCKET;
    }

    return s;
}



/***************************************************************************
 *
 > $Function: commandInstrument$
 *
 * $Description:  send a SCPI command to the instrument.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 * $Return:  (char *) . . . . . . a pointer to the result string.
 *
 * $Errors:   returns 0 if send fails $
 *
 ***************************************************************************/
int commandInstrument(SOCKET sock,
                      const char *command)
{
```


Programming Examples

```
    int count;

    /* fprintf(stderr, "Sending \"%s\".\n", command);  */
    if (strchr(command, '\n') == NULL) {
       fprintf(stderr, "Warning: missing newline on command %s.\n", command);
    }

    count = send(sock, command, strlen(command), 0);
    if (count == SOCKET_ERROR) {
        return COMMAND_ERROR;
    }

    return NO_CMD_ERROR;
}


/***************************************************************************
 * recv_line(): similar to fgets(), but uses recv()
 **************************************************************************/
char * recv_line(SOCKET sock, char * result, int maxLength)
{
#ifdef WINSOCK
    int cur_length = 0;
    int count;
    char * ptr = result;
    int err = 1;

    while (cur_length < maxLength) {
        /* Get a byte into ptr */
        count = recv(sock, ptr, 1, 0);

        /* If no chars to read, stop. */
        if (count < 1) {
            break;
        }
        cur_length += count;
```

Programming Examples

```
        /* If we hit a newline, stop. */
        if (*ptr == '\n') {
            ptr++;
            err = 0;
            break;
        }
        ptr++;

    }

    *ptr = '\0';

    if (err) {
        return NULL;
    } else {
        return result;
    }
#else
    /*********************************************************************
     * Simpler UNIX version, using file I/O.  recv() version works too.
     * This demonstrates how to use file I/O on sockets, in UNIX.
     *********************************************************************/
    FILE * instFile;
    instFile = fdopen(sock, "r+");
    if (instFile == NULL)
    {
        fprintf(stderr, "Unable to create FILE * structure : %s\n",
                strerror(errno));
        exit(2);
    }
    return fgets(result, maxLength, instFile);
#endif
}
```

Programming Examples

```
/***************************************************************************
 *
 > $Function: queryInstrument$
 *
 * $Description:  send a SCPI command to the instrument, return a response.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 *     (char *result) . . . . . . where to put the result.
 *     (size_t) maxLength . . . . maximum size of result array in bytes.
 *
 * $Return:  (long) . . . . . . . The number of bytes in result buffer.
 *
 * $Errors:   returns 0 if anything goes wrong. $
 *
 ***************************************************************************/
long queryInstrument(SOCKET sock,
                      const char *command, char *result, size_t maxLength)
{
    long ch;
    char tmp_buf[8];
    long resultBytes = 0;
    int command_err;
    int count;

    /*********************************************************
     * Send command to analyzer
     *********************************************************/
    command_err = commandInstrument(sock, command);
    if (command_err) return COMMAND_ERROR;


    /*********************************************************
     * Read response from analyzer
```

Programming Examples

```
 *****************************************************/
count = recv(sock, tmp_buf, 1, 0); /* read 1 char */
ch = tmp_buf[0];

if ((count < 1) || (ch == EOF)  || (ch == '\n'))
{
    *result = '\0';  /* null terminate result for ascii */
    return 0;
}

/* use a do-while so we can break out */
do
{
    if (ch == '#')
    {
        /* binary data encountered - figure out what it is */
        long numDigits;
        long numBytes = 0;
        /* char length[10]; */

        count = recv(sock, tmp_buf, 1, 0); /* read 1 char */
        ch = tmp_buf[0];
        if ((count < 1) || (ch == EOF)) break; /* End of file */

        if (ch < '0' || ch > '9') break;  /* unexpected char */
        numDigits = ch - '0';

        if (numDigits)
        {
            /* read numDigits bytes into result string. */
            count = recv(sock, result, (int)numDigits, 0);
            result[count] = 0;  /* null terminate */
            numBytes = atol(result);
        }

        if (numBytes)
```

```
    {
        resultBytes = 0;
        /* Loop until we get all the bytes we requested. */
        /* Each call seems to return up to 1457 bytes, on HP-UX 9.05 */
        do {
            int rcount;
            rcount = recv(sock, result, (int)numBytes, 0);
            resultBytes += rcount;
            result      += rcount;  /* Advance pointer */
        } while ( resultBytes < numBytes );

        /************************************************************
         * For LAN dumps, there is always an extra trailing newline
         * Since there is no EOI line.  For ASCII dumps this is
         * great but for binary dumps, it is not needed.
         ************************************************************/
        if (resultBytes == numBytes)
        {
            char junk;
            count = recv(sock, &junk, 1, 0);
        }
    }
    else
    {
        /* indefinite block ... dump til we read only a line feed */
        do
        {
            if (recv_line(sock, result, maxLength) == NULL) break;
            if (strlen(result)==1 && *result == '\n') break;
            resultBytes += strlen(result);
            result += strlen(result);
        } while (1);
    }
}
else
{
```

Programming Examples

```
            /* ASCII response (not a binary block) */
            *result = (char)ch;
            if (recv_line(sock, result+1, maxLength-1) == NULL) return 0;

            /* REMOVE trailing newline, if present.  And terminate string. */
            resultBytes = strlen(result);
            if (result[resultBytes-1] == '\n') resultBytes -= 1;
            result[resultBytes] = '\0';
        }
    } while (0);

    return resultBytes;
}



/*************************************************************************
 *
 > $Function: showErrors$
 *
 * $Description: Query the SCPI error queue, until empty.  Print results. $
 *
 * $Return:  (void)
 *
 *************************************************************************/
void showErrors(SOCKET sock)
{
    const char * command = "SYST:ERR?\n";
    char result_str[256];

    do {
        queryInstrument(sock, command, result_str, sizeof(result_str)-1);

        /******************************************************************
         * Typical result_str:
```

```
        *     -221,"Settings conflict; Frequency span reduced."
        *     +0,"No error"
        * Don't bother decoding.
        ******************************************************************/
        if (strncmp(result_str, "+0,", 3) == 0) {
            /* Matched +0,"No error" */
            break;
        }
        puts(result_str);
    } while (1);

}


/****************************************************************************
 *
 > $Function: isQuery$
 *
 * $Description: Test current SCPI command to see if it a query. $
 *
 * $Return:  (unsigned char) . . . non-zero if command is a query.  0 if not.
 *
 ****************************************************************************/
unsigned char isQuery( char* cmd )
{
    unsigned char q = 0 ;
    char *query ;

    /*********************************************************/
    /* if the command has a '?' in it, use queryInstrument.  */
    /* otherwise, simply send the command.                   */
    /* Actually, we must a little more specific so that      */
    /* marker value queries are treated as commands.         */
    /* Example:  SENS:FREQ:CENT (CALC1:MARK1:X?)             */
    /*********************************************************/
    if ( (query = strchr(cmd,'?')) != NULL)
```

Programming Examples

```
    {
        /* Make sure we don’t have a marker value query, or
         * any command with a ’?’ followed by a ’)’ character.
         * This kind of command is not a query from our point of view.
         * The analyzer does the query internally, and uses the result.
         */
        query++ ;       /* bump past ’?’ */
        while (*query)
        {
            if (*query == ’ ’) /* attempt to ignore white spc */
                query++ ;
            else break ;
        }

        if ( *query != ’)’ )
        {
            q = 1 ;
        }
    }
    return q ;
}




/***************************************************************************
 *
 > $Function: main$
 *
 * $Description: Read command line arguments, and talk to analyzer.
                 Send query results to stdout. $
 *
 * $Return:  (int) . . . non-zero if an error occurs
 *
 ***************************************************************************/
int main(int argc, char *argv[])
{
```

```
SOCKET instSock;
char *charBuf = (char *) malloc(INPUT_BUF_SIZE);
char *basename;
int chr;
char command[1024];
char *destination;
unsigned char quiet = 0;
unsigned char show_errs = 0;
int number = 0;

basename = strrchr(argv[0], '/');
if (basename != NULL)
    basename++ ;
else
    basename = argv[0];

while ( ( chr = getopt(argc,argv,"qune")) != EOF )
    switch (chr)
    {
        case 'q':  quiet = 1; break;
        case 'n':  number = 1; break ;
        case 'e':  show_errs = 1; break ;
        case 'u':
        case '?':  usage(basename); exit(1) ;
    }

/* now look for hostname and optional <command> */
if (optind < argc)
{
    destination = argv[optind++] ;
    strcpy(command, "");
    if (optind < argc)
    {
        while (optind < argc) {
            /* <hostname> <command> provided; only one command string */
```

Programming Examples

```
                strcat(command, argv[optind++]);
                if (optind < argc) {
                    strcat(command, " ");
                } else {
                    strcat(command, "\n");
                }
            }
        }
        else
        {
            /* Only <hostname> provided; input on <stdin> */
            strcpy(command, "");

            if (optind > argc)
            {
                usage(basename);
                exit(1);
            }
        }
    }
    else
    {
        /* no hostname! */
        usage(basename);
        exit(1);
    }

    /**********************************************/
    /* open a socket connection to the instrument */
    /**********************************************/
#ifdef WINSOCK
    if (init_winsock() != 0) {
        exit(1);
    }
#endif /* WINSOCK */
```


Programming Examples

```
instSock = openSocket(destination, SCPI_PORT);
if (instSock == INVALID_SOCKET) {
    fprintf(stderr, "Unable to open socket.\n");
    return 1;
}
/* fprintf(stderr, "Socket opened.\n"); */

if (strlen(command) > 0)
{
    /*******************************************************/
    /* if the command has a '?' in it, use queryInstrument. */
    /* otherwise, simply send the command.                 */
    /*******************************************************/
    if ( isQuery(command) )
    {
        long bufBytes;
        bufBytes = queryInstrument(instSock, command,
                                   charBuf, INPUT_BUF_SIZE);
        if (!quiet)
        {
            fwrite(charBuf, bufBytes, 1, stdout);
            fwrite("\n", 1, 1, stdout) ;
            fflush(stdout);
        }
    }
    else
    {
        commandInstrument(instSock, command);
    }
}
else
{
    /* read a line from <stdin> */
    while ( gets(charBuf) != NULL )
    {
        if ( !strlen(charBuf) )
```

```
        continue ;

    if ( *charBuf == '#' || *charBuf == '!' )
        continue ;

    strcat(charBuf, "\n");

    if (!quiet)
    {
        if (number)
        {
            char num[10];
            sprintf(num,"%d: ",number);
            fwrite(num, strlen(num), 1, stdout);
        }
        fwrite(charBuf, strlen(charBuf), 1, stdout) ;
        fflush(stdout);
    }

    if ( isQuery(charBuf) )
    {
        long bufBytes;

        /* Put the query response into the same buffer as the
         * command string appended after the null terminator.
         */
        bufBytes = queryInstrument(instSock, charBuf,
                                   charBuf + strlen(charBuf) + 1,
                                   INPUT_BUF_SIZE -strlen(charBuf) );
        if (!quiet)
        {
            fwrite("  ", 2, 1, stdout) ;
            fwrite(charBuf + strlen(charBuf)+1, bufBytes, 1, stdout);
            fwrite("\n", 1, 1, stdout) ;
            fflush(stdout);
        }
```


Programming Examples

```
            }
            else
            {
                commandInstrument(instSock, charBuf);
            }
            if (number) number++;
        }
    }

    if (show_errs) {
        showErrors(instSock);
    }

#ifdef WINSOCK
    closesocket(instSock);
    close_winsock();
#else
    close(instSock);
#endif /* WINSOCK */

    return 0;
}

/* End of lanio.c */
```

Programming Examples

Using C Over Socket LAN (Windows NT)

This C programming example (getopt.c) compiles in the Windows NT environment. In Windows, the routines send( ) and recv( ) must be used, since fread( ) and fwrite( ) may not work on sockets.

The program reads the analyzer's host name from the command line, followed by the SCPI command. It then opens a socket to the analyzer, using port 5025, and sends the command. If the command appears to be a query, the program queries the analyzer for a response, and prints the response.

This example program can also be used as a utility to talk to your analyzer from the command prompt on your Windows NT PC, or from within a script.

Example:

```
/***************************************************************************

 getopt(3C)                                                        getopt(3C)



 NAME
      getopt - get option letter from argument vector

 SYNOPSIS
      int getopt(int argc, char * const argv[], const char *optstring);

      extern char *optarg;
      extern int optind, opterr, optopt;

 DESCRIPTION
      getopt returns the next option letter in argv (starting from argv[1])
      that matches a letter in optstring.  optstring is a string of
      recognized option letters; if a letter is followed by a colon, the
      option is expected to have an argument that may or may not be
      separated from it by white space.  optarg is set to point to the start
      of the option argument on return from getopt.
```

```
    getopt places in optind the argv index of the next argument to be
    processed.  The external variable optind is initialized to 1 before
    the first call to the function getopt.

    When all options have been processed (i.e., up to the first non-option
    argument), getopt returns EOF.  The special option -- can be used to
    delimit the end of the options; EOF is returned, and -- is skipped.

 ***************************************************************************/


#include <stdio.h>      /* For NULL, EOF */
#include <string.h>     /* For strchr() */

char    *optarg;        /* Global argument pointer. */
int     optind = 0;     /* Global argv index. */

static char     *scan = NULL;   /* Private scan pointer. */

int getopt( int argc, char * const argv[], const char* optstring)
{
    char c;
    char *posn;

    optarg = NULL;

    if (scan == NULL || *scan == '\0') {
        if (optind == 0)
            optind++;

        if (optind >= argc || argv[optind][0] != '-' || argv[optind][1] == '\0')
            return(EOF);
        if (strcmp(argv[optind], "--")==0) {
            optind++;
            return(EOF);
        }
```


Programming Examples

```
        scan = argv[optind]+1;
        optind++;
    }

    c = *scan++;
    posn = strchr(optstring, c);        /* DDP */

    if (posn == NULL || c == ':') {
        fprintf(stderr, "%s: unknown option -%c\n", argv[0], c);
        return('?');
    }

    posn++;
    if (*posn == ':') {
        if (*scan != '\0') {
            optarg = scan;
            scan = NULL;
        } else {
            optarg = argv[optind];
            optind++;
        }
    }

    return(c);
}
```

Using Java Programming Over Socket LAN

This Java programming example (ScpiDemo.java) demonstrates simple socket programming with Java and can be found on the Documentation CD. It is written in Java programming language, and will compile with Java compilers versions 1.0 and above.

Example:

```
import java.awt.*;
import java.io.*;
import java.net.*;
import java.applet.*;

// This is a SCPI Demo to demonstrate how one can communicate with the
// E4440A PSA  with a JAVA capable browser.  This is the
// Main class for the SCPI Demo.  This applet will need Socks.class to
// support the I/O commands and a ScpiDemo.html for a browser to load
// the applet.
// To use this applet, either compile this applet with a Java compiler
// or use the existing compiled classes.  copy ScpiDemo.class,
// Socks.class and ScpiDemo.html to a floppy.  Insert the floppy into
// your instrument.  Load up a browser on your computer and do the
// following:
//     1. Load this URL in your browser:
//        ftp://<Your instrument's IP address or name>/int/ScpiDemo.html
//     2. There should be two text windows show up in the browser:
//        The top one is the SCPI response text area for any response
//        coming back from the instrument.  The bottom one is for you
//        to enter a SCPI command.  Type in a SCPI command and hit enter.
//        If the command expects a response, it will show up in the top
//        window.
public class ScpiDemo extends java.applet.Applet implements Runnable {
    Thread       responseThread;
    Socks        sck;
    URL          appletBase;
    TextField    scpiCommand = new TextField();
```

Programming Examples

```
TextArea      scpiResponse = new TextArea(10, 60);
Panel         southPanel = new Panel();
Panel         p;

// Initialize the applets
public void init() {

    SetupSockets();
    SetupPanels();

    // Set up font type for both panels
    Font font = new Font("TimesRoman", Font.BOLD,14);
    scpiResponse.setFont(font);
    scpiCommand.setFont(font);
    scpiResponse.appendText("SCPI Demo Program:  Response messages\n");
    scpiResponse.appendText("--------------------------------------------\n");
}

// This routine is called whenever the applet is actived
public void start() {
    // Open the sockets if not already opened
    sck.OpenSockets();
    // Start a response thread
    StartResponseThread(true);
}

// This routine is called whenever the applet is out of scope
// i.e. minize browser
public void stop() {
    // Close all local sockets
    sck.CloseSockets();
    // Kill the response thread
    StartResponseThread(false);
}

// Action for sending out scpi commands
```

```
// This routine is called whenever a command is received from the
// SCPI command panel.
public boolean action(Event evt, Object what) {
    // If this is the correct target
    if (evt.target == scpiCommand) {
        // Get the scpi command
        String str = scpiCommand.getText();
        // Send it out to the Scpi socket
        sck.ScpiWriteLine(str);
        String tempStr = str.toLowerCase();
        // If command str is "syst:err?", don't need to send another one.
        if ( (tempStr.indexOf("syst") == -1) ||
             (tempStr.indexOf("err") == -1)  ) {
            // Query for any error
            sck.ScpiWriteLine("syst:err?");
        }
        return true;
    }
    return false;
}

// Start/Stop a Response thread to display the response strings
private void StartResponseThread(boolean start) {
    if (start) {
        // Start a response thread
        responseThread = new Thread(this);
        responseThread.start();
    }
    else {
        // Kill the response thread
        responseThread = null;
    }
}

// Response thread running
public void run() {
```

Programming Examples

```
        String str = "";  // Initialize str to null

        // Clear the error queue before starting the thread
        // in case if there’s any error messages from the previous actions
        while ( str.indexOf("No error") == -1 ) {
            sck.ScpiWriteLine("syst:err?");
            str = sck.ScpiReadLine();
        }

        // Start receiving response or error messages
        while(true) {
            str = sck.ScpiReadLine();
            if ( str != null ) {
                // If response messages is "No error", do no display it,
                // replace it with "OK" instead.
                if ( str.equals("+0,\"No error\"") ) {
                    str = "OK";
                }
                // Display any response messages in the Response panel
                scpiResponse.appendText(str+"\n");
            }
        }
    }

    // Set up and open the SCPI sockets
    private void SetupSockets() {
        // Get server url
        appletBase = (URL)getCodeBase();
        // Open the sockets
        sck = new Socks(appletBase);
    }

    // Set up the SCPI command and response panels
    private void SetupPanels() {
        // Set up SCPI command panel
        southPanel.setLayout(new GridLayout(1, 1));
```

```
        p = new Panel();
        p.setLayout(new BorderLayout());
        p.add("West", new Label("SCPI command:"));
        p.add("Center", scpiCommand);
        southPanel.add(p);

        // Set up the Response panel
        setLayout(new BorderLayout(2,2));
        add("Center", scpiResponse);
        add("South", southPanel);
    }

}



//  Socks class is responsible for open/close/read/write operations
//  from the predefined socket ports.  For this example program,
//  the only port used is 5025 for the SCPI port.
class Socks extends java.applet.Applet {
    // Socket Info
    // To add a new socket, add a constant here, change MAX_NUM_OF_SOCKETS
    // then, edit the constructor for the new socket.
    public final int SCPI=0;
    private final int MAX_NUM_OF_SOCKETS=1;

    // Port number
    // 5025 is the dedicated port number for E4440A Scpi Port
    private final int SCPI_PORT = 5025;

    // Socket info
    private URL appletBase;
    private Socket[] sock = new Socket[MAX_NUM_OF_SOCKETS];
    private DataInputStream[] sockIn = new DataInputStream[MAX_NUM_OF_SOCKETS];
    private PrintStream[] sockOut = new PrintStream[MAX_NUM_OF_SOCKETS];
    private int[] port = new int[MAX_NUM_OF_SOCKETS];
    private boolean[] sockOpen = new boolean[MAX_NUM_OF_SOCKETS];
```

```
// Constructor
Socks(URL appletB)
{
    appletBase = appletB;

    // Set up for port array.
    port[SCPI] = SCPI_PORT;

    // Initialize the sock array
    for ( int i = 0; i < MAX_NUM_OF_SOCKETS; i++ ) {
        sock[i] = null;
        sockIn[i] = null;
        sockOut[i] = null;
        sockOpen[i] = false;
    }
}

//***** Sockects open/close routines
// Open the socket(s) if not already opened
public void OpenSockets()
{
    try {
        // Open each socket if possible
        for ( int i = 0; i < MAX_NUM_OF_SOCKETS; i++ ) {
            if ( !sockOpen[i] ) {
                sock[i] = new Socket(appletBase.getHost(),port[i]);
                sockIn[i] = new DataInputStream(sock[i].getInputStream());
                sockOut[i] = new PrintStream(sock[i].getOutputStream());
                if ( (sock[i] != null) && (sockIn[i] != null) &&
                     (sockOut[i] != null) ) {
                    sockOpen[i] = true;
                }
            }
        }
    }
```

```
    catch (IOException e) {
        System.out.println("Sock, Open Error "+e.getMessage());
    }
}

// Close the socket(s) if opened
public void CloseSocket(int s)
{
    try {
        if ( sockOpen[s] == true ) {
            // write blank line to exit servers elegantly
            sockOut[s].println();
            sockOut[s].flush();
            sockIn[s].close();
            sockOut[s].close();
            sock[s].close();
            sockOpen[s] = false;
        }
    }
    catch (IOException e) {
        System.out.println("Sock, Close Error "+e.getMessage());
    }
}

// Close all sockets
public void CloseSockets()
{
    for ( int i=0; i < MAX_NUM_OF_SOCKETS; i++ ) {
        CloseSocket(i);
    }
}

// Return the status of the socket, open or close.
public boolean SockOpen(int s)
{
    return sockOpen[s];
```

```
}


//************* Socket I/O routines.

//*** I/O routines for SCPI socket

// Write an ASCII string with carriage return to SCPI socket
public void ScpiWriteLine(String command)
{
    if ( SockOpen(SCPI) ) {
        sockOut[SCPI].println(command);
        sockOut[SCPI].flush();
    }
}

// Read an ASCII string, terminated with carriage return from SCPI socket
public String ScpiReadLine()
{
    try {
        if ( SockOpen(SCPI) ) {
            return sockIn[SCPI].readLine();
        }
    }
    catch (IOException e) {
        System.out.println("Scpi Read Line Error "+e.getMessage());
    }
    return null;
}

// Read a byte from SCPI socket
public byte ScpiReadByte()
{
    try {
        if ( SockOpen(SCPI) ) {
            return sockIn[SCPI].readByte();
```

```
            }
        }
        catch (IOException e) {
            System.out.println("Scpi Read Byte Error "+e.getMessage());
        }
        return 0;
    }

}
```

Using the VXI Plug-N-Play Driver in LabVIEW®

This example shows how to use the VXI plug and play driver over LAN in LabVIEW 6. The vi file (lan_pnp.vi) can be found on the Documentation CD.

You must have Version K of the Agilent IO libraries installed on your PC, either alone or installed side-by-side with the National Instruments IO libraries. Also, you must first import the VXI plug and play driver into LabVIEW before running this example. The instrument drivers are available at:

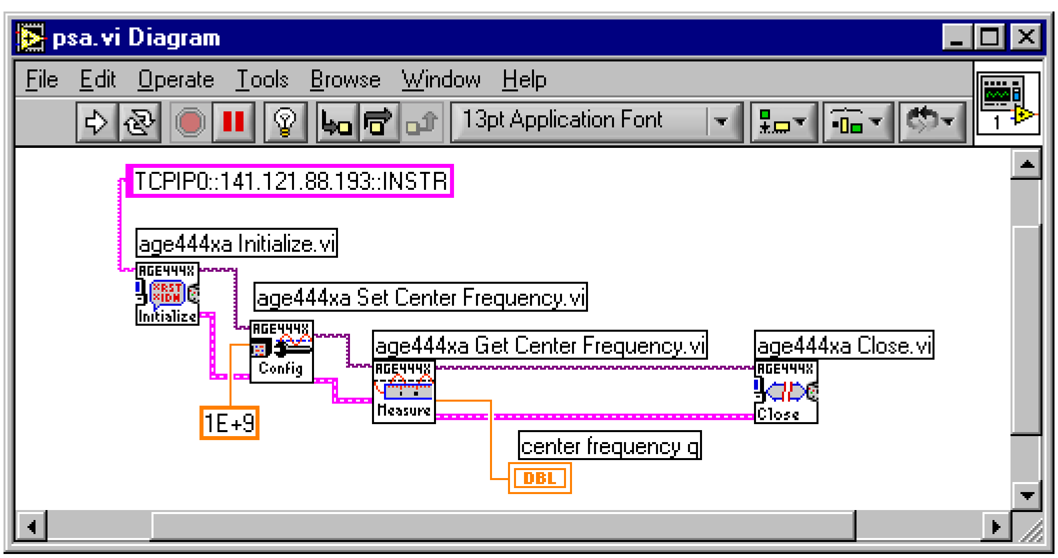
`http://www.agilent.com/find/iolib` (Click on instrument drivers.)

This example:

1. Opens a VXI 11.3 Lan connection to the instrument
2. Sets the Center Frequency to 1 GHz
3. Queries the instrument's center frequency
4. Closes the Lan connection to the instrument

NOTE Substitute your instruments I.P. address for the one used in the example.

Example:



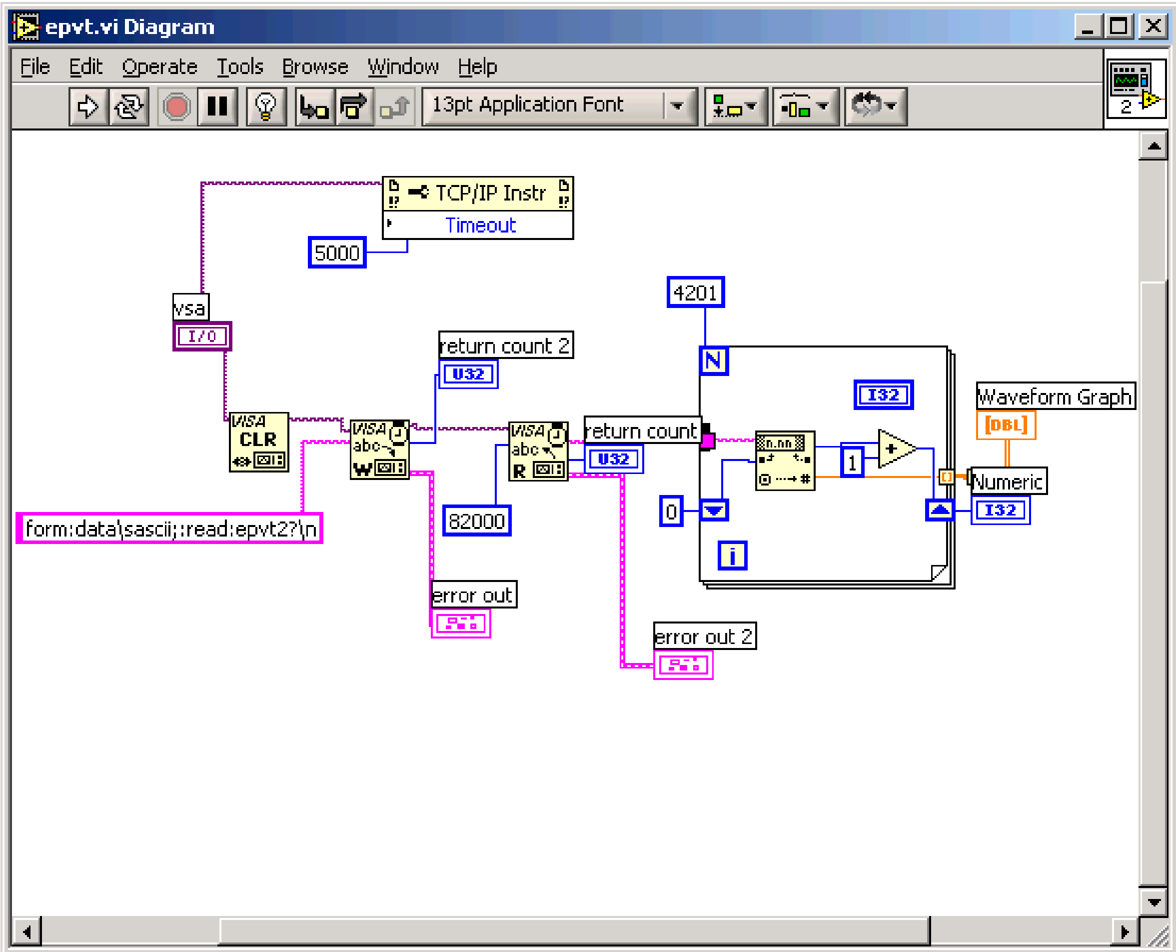
Using LabVIEW® 6 to Make an EDGE GSM Measurement

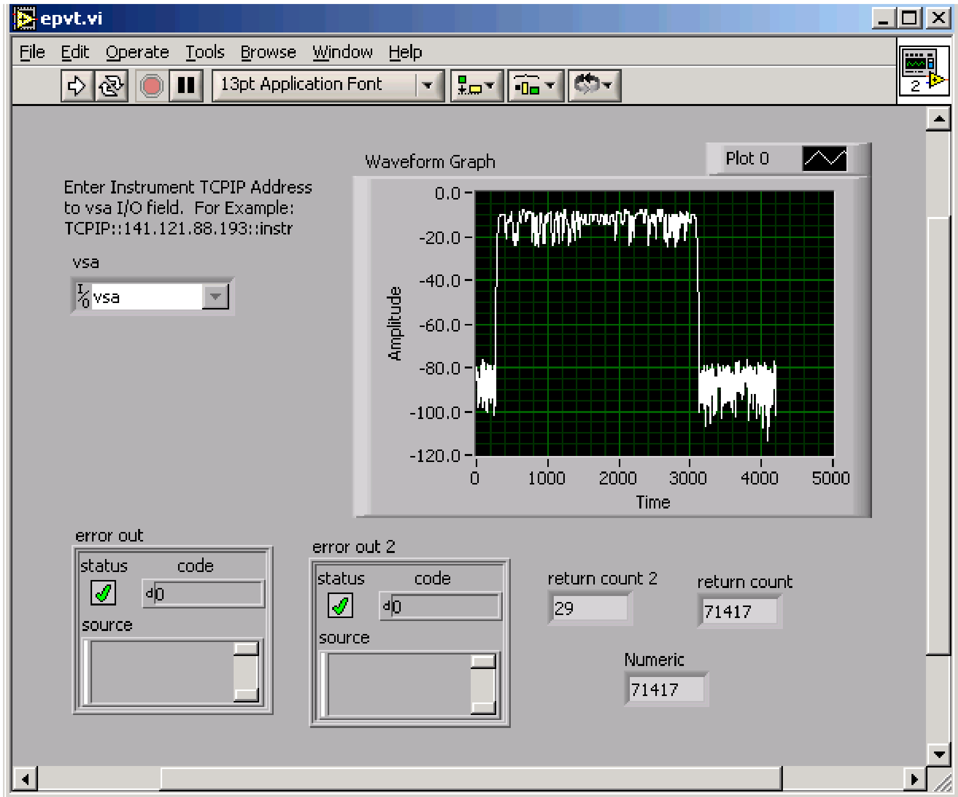
This is a LabVIEW 6 example that uses SCPI commands instead of the instrument driver. It demonstrates reading ASCII trace points of entire EDGE waveform data in the Power Vs. Time measurement over LAN. This program uses the optional GSM/EDGE personality in the PSA Series Spectrum Analyzers and in the E4406A Vector Signal Analyzer. The vi file (epvt.vi) can be found on the Documentation CD.

This example:

1. Opens a VXI 11.3 Lan connection to the instrument
2. Changes the data format to ASCII.
3. Initiates a power vs. time measurement and reads the results.
4. The comma separated ASCII results string is converted to an array of values.

Example:


epvt.vi Diagram
File Edit Operate Tools Browse Window Help
13pt Application Font
TCP/IP Instr
Timeout
5000
vsa
I/O
VISA CLR
VISA abc W
VISA abc R
return count 2
U32
return count
U32
form:data\sascii;:read:epvt2?\n
82000
error out
error out 2
4201
N
I32
0
i
1
Waveform Graph
DBL
Numeric
I32


epvt.vi
File Edit Operate Tools Browse Window Help
13pt Application Font
Enter Instrument TCPIP Address
to vsa I/O field. For Example:
TCPIP::141.121.88.193::instr
vsa
vsa
Waveform Graph
Plot 0
Amplitude
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
-120.0
0
1000
2000
3000
4000
5000
Time
error out
status
code
0
source
error out 2
status
code
0
source
return count 2
29
return count
71417
Numeric
71417

Using Visual Basic® 6 to Capture a Screen Image

This is a Visual Basic example that stores the current screen image on your PC. The program works with the ESA or PSA Series spectrum analyzers. The bas file (screen.bas) and a compiled executable (screen.exe) can be found on the Documentation CD.

This example:

1. Stores the current screen image on the instrument's flash as C:PICTURE.GIF.
2. Transfers the image over GPIB or LAN and stores it on your PC in the current directory as picture.gif.
3. The file C:PICTURE.GIF is then deleted from the instrument's flash.

NOTE This example uses GPIB address 18 for the spectrum analyzer.

```
'' """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
''  Copyright (c) 1999- 2003 Agilent Technologies Inc.  All rights reserved.
''
'' You have a royalty-free right to use, modify, reproduce and distribute
'' the Sample Application Files (and/or any modified version) in any way
'' you find useful, provided that you agree that Agilent Technologies has
'' no warranty, obligations or liability for any Sample Application Files.
''
'' Agilent Technologies provides programming examples for illustration only,
'' This sample program assumes that you are familiar with the programming
'' language being demonstrated and the tools used to create and debug
'' procedures. Agilent Technologies support engineers can help explain the
'' functionality of Agilent Technologies software components and associated
'' commands, but they will not modify these samples to provide added
'' functionality or construct procedures to meet your specific needs.
'' """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

'' To develop VISA applications in Microsoft Visual Basic, you first need
'' to add the Visual Basic (VB) declaration file in your VB project as a
'' Module. This file contains the VISA function definitions and constant
```

```
'' declarations needed to make VISA calls from Visual Basic.
'' To add this module to your project in VB 6, from the menu, select
'' Project->Add Module, select the 'Existing' tab, and browse to the
'' directory containing the VB Declaration file, select visa32.bas, and
'' press 'Open'.
''
'' The name and location of the VB declaration file depends on which
'' operating system you are using.  Assuming the 'standard' VISA directory
'' of C:\Program Files\VISA or the 'standard' VXIpnp directory of
'' C:\VXIpnp, the visa32.bas file can be located in one of the following:
''
''   \winnt\agvisa\include\visa32.bas - Windows NT/2000/XP
''   \winnt\include\visa32.bas        - Windows NT/2000/XP
''   \win95\include\visa32.bas        - Windows 95/98/Me
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'  screen.bas
'  The following example program is written for the PSA and ESA Series
'  Spectrum Analyzers.  It stores the current screen image on the
'  instrument's flash as C:PICTURE.GIF. It then transfers the image over
'  GPIB or LAN and stores the image on your PC in the current directory
'  as picture.gif.  The file C:PICTURE.GIF is then deleted on the
'  instrument's flash.
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Option Explicit

Private Sub Main()
    ' Declare Variables used in the program
    Dim status As Long       'VISA function status return code
    Dim defrm As Long        'Session to Default Resource Manager
    Dim vi As Long           'Session to instrument
    Dim x As Integer         'Loop Variable
    Dim ArrayPtr(1) As Long 'Array of Pointers
    Dim ResultsArray(50000) As Byte 'results array, Big enough to hold a GIF
    Dim length As Long       'Number of bytes returned from instrument
    Dim fnum As Integer      'File Number to used to open file to store data
    Dim isOpen As Boolean    'Boolean flag used to keep track of open file
```

Programming Examples

```
Dim headerlength As Long 'length of header

'Set the default number of bytes that will be contained in the
'ResultsArray to 50,000 (50kB)
length = 50000

'Set the array of pointers to the addresses of the variables
ArrayPtr(0) = VarPtr(length)
ArrayPtr(1) = VarPtr(ResultsArray(0))

'Delete picture.gif file if it exists
On Error Resume Next
Kill "picture.gif"

On Error GoTo Error_Handler

' Open the default resource manager session
status = viOpenDefaultRM(defrm)

' Open the session.  Note: For PSA, to use LAN, change the string to
' "TCPIP0::xxx.xxx.xxx.xxx::inst0::INSTR" where xxxxx is the IP address
status = viOpen(defrm, "GPIB0::18::INSTR", 0, 0, vi)
If (status < 0) Then GoTo VisaErrorHandler

' Set the I/O timeout to fifteen seconds
status = viSetAttribute(vi, VI_ATTR_TMO_VALUE, 15000)
If (status < 0) Then GoTo VisaErrorHandler

'Store the current screen image on flash as C:PICTURE.GIF
status = viVPrintf(vi, ":MMEM:STOR:SCR 'C:PICTURE.GIF'" + Chr$(10), 0)
If (status < 0) Then GoTo VisaErrorHandler

'Grab the screen image file from the instrument
status = viVQueryf(vi, ":MMEM:DATA? 'C:PICTURE.GIF'" + Chr$(10), _
         "%#y", ArrayPtr(0))
```

Programming Examples

```
    'Delete the tempory file on the flash named C:PICTURE.GIF
    status = viVPrintf(vi, ":MMEM:DEL 'C:PICTURE.GIF'" + Chr$(10), 0)
    If (status < 0) Then GoTo VisaErrorHandler

    'Close the vi session and the resource manager session
    Call viClose(vi)
    Call viClose(defrm)

    'Store the results in a text file
    fnum = FreeFile()  'Get the next free file number
    Open "picture.gif" For Binary As #fnum
    isOpen = True
    headerlength = 2 + (Chr$(ResultsArray(1)))
    For x = headerlength To length - 2
    Put #fnum, , ResultsArray(x)
    Next x

    ' Intentionally flow into Error Handler to close file
Error_Handler:
    ' Raise the error (if any), but first close the file
    If isOpen Then Close #fnum
    If Err Then Err.Raise Err.Number, , Err.Description
    Exit Sub

VisaErrorHandler:
   Dim strVisaErr As String * 200
   Call viStatusDesc(defrm, status, strVisaErr)
   MsgBox "*** Error : " & strVisaErr, vbExclamation, "VISA Error Message"
   Exit Sub
End Sub
```

Using Visual Basic® 6 to Transfer Binary Trace Data

This is a Visual Basic example that gets binary trace data from the instrument. Binary data transfers are faster than the default ASCII transfer mode, because less data is sent over the bus. This example works with the ESA or PSA Series spectrum analyzers. The bas file (bintrace.bas) and a compiled executable (bintrace.exe) can be found on the Documentation CD.

This example:

1. Queries the IDN (identification) string from the instrument.
2. While in Spectrum Analysis mode, it reads the trace data in binary format (Real,32 or Real,64 or Int,32).
3. Stores the data is then to a file "bintrace.txt".

NOTE This example uses GPIB address 18 for the spectrum analyzer.

```
'' """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
''  Copyright (c) 1999- 2003 Agilent Technologies Inc.  All rights reserved.
''
'' You have a royalty-free right to use, modify, reproduce and distribute
'' the Sample Application Files (and/or any modified version) in any way
'' you find useful, provided that you agree that Agilent Technologies has
'' no warranty, obligations or liability for any Sample Application Files.
''
'' Agilent Technologies provides programming examples for illustration only,
'' This sample program assumes that you are familiar with the programming
'' language being demonstrated and the tools used to create and debug
'' procedures. Agilent Technologies support engineers can help explain the
'' functionality of Agilent Technologies software components and associated
'' commands, but they will not modify these samples to provide added
'' functionality or construct procedures to meet your specific needs.
'' """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

'' To develop VISA applications in Microsoft Visual Basic, you first need
'' to add the Visual Basic (VB) declaration file in your VB project as a
'' Module. This file contains the VISA function definitions and constant
```

```
'' declarations needed to make VISA calls from Visual Basic.
'' To add this module to your project in VB 6, from the menu, select
'' Project->Add Module, select the 'Existing' tab, and browse to the
'' directory containing the VB Declaration file, select visa32.bas, and
'' press 'Open'.
''
'' The name and location of the VB declaration file depends on which
'' operating system you are using.  Assuming the 'standard' VISA directory
'' of C:\Program Files\VISA or the 'standard' VXIpnp directory of
'' C:\VXIpnp, the visa32.bas file can be located in one of the following:
''
''   \winnt\agvisa\include\visa32.bas - Windows NT/2000/XP
''   \winnt\include\visa32.bas        - Windows NT/2000/XP
''   \win95\include\visa32.bas        - Windows 95/98/Me

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'  bintrace.bas
'  The following example program is written for the PSA and ESA Series
'  Spectrum Analyzers.  It queries the IDN string from the instrument
'  and then reads the trace data in Spectrum Analysis mode in binary
'  format (Real,32 or Real,64 or Int,32).  The data is then stored to a
'  file "bintrace.txt".
'  Binary transfers are faster than the default ASCII transfer mode,
'  because less data is sent over the bus.
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Option Explicit

Private Sub Main()
    ' Declare Variables used in the program
    Dim status As Long       'VISA function status return code
    Dim defrm As Long        'Session to Default Resource Manager
    Dim vi As Long           'Session to instrument
    Dim strRes As String * 100  'Fixed length string to hold *IDN? Results
    Dim x As Integer         'Loop Variable
    Dim output As String     'output string variable
    Dim ArrayPtr(1) As Long 'Array of Pointers
```

```
Dim ResultsArray(8192) As Single 'trace element array of Real,32 values
'For Real,64 data use Double. For Int,32 data use Long
Dim length As Long      'Number of trace elements return from instrument
Dim fnum As Integer     'File Number to used to open file to store data
Dim isOpen As Boolean   'Boolean flag used to keep track of open file

'Set the default number of trace elements to the ResultsArray size
'Note: PSA and ESA currently support up to 8192 trace points
length = 8192

'Set the array of pointers to the addresses of the variables
ArrayPtr(0) = VarPtr(length)
ArrayPtr(1) = VarPtr(ResultsArray(0))

On Error GoTo Error_Handler

' Open the default resource manager session
status = viOpenDefaultRM(defrm)

' Open the session.  Note: For PSA, to use LAN, change the string to
' "TCPIP0::xxx.xxx.xxx.xxx::inst0::INTSR" where xxxxx is the IP address
status = viOpen(defrm, "GPIB0::18::INSTR", 0, 0, vi)
If (status < 0) Then GoTo VisaErrorHandler

' Set the I/O timeout to five seconds
status = viSetAttribute(vi, VI_ATTR_TMO_VALUE, 5000)
If (status < 0) Then GoTo VisaErrorHandler

'Ask for the devices's *IDN string.
status = viVPrintf(vi, "*IDN?" + Chr$(10), 0)
If (status < 0) Then GoTo VisaErrorHandler

'Read back the IDN string from the instrument
status = viVScanf(vi, "%t", strRes)
If (status < 0) Then GoTo VisaErrorHandler
```

Programming Examples

```
'Print the IDN string results in a message box
MsgBox (strRes)

'Change the instrument mode to Spectrum Analysis
status = viVPrintf(vi, ":INST:NSEL 1" + Chr$(10), 0)
If (status < 0) Then GoTo VisaErrorHandler

' Set instrument trace data format to 32-bit Real
' Note: For higher precision use 64-bit data, ":FORM REAL,64"
' For faster data transfer for ESA, use ":FORM INT,32"
status = viVPrintf(vi, ":FORM REAL,32" + Chr$(10), 0)
If (status < 0) Then GoTo VisaErrorHandler

'Set Analyzer to single sweep mode
status = viVPrintf(vi, ":INIT:CONT 0" + Chr$(10), 0)
If (status < 0) Then GoTo VisaErrorHandler

'Trigger a sweep and wait for sweep to complete
status = viVPrintf(vi, ":INIT:IMM;*WAI" + Chr$(10), 0)
If (status < 0) Then GoTo VisaErrorHandler

'Query the trace data from the instrument
'Note: Change the "%#zb" to "%#Zb" for Real,64 data
'      For Int,32 leave the modifier as "%#zb"
status = viVQueryf(vi, ":TRAC:DATA? TRACE1" + Chr$(10), _
         "%#zb", ArrayPtr(0))

'Close the vi session and the resource manager session
Call viClose(vi)
Call viClose(defrm)

'Print number of elements returned
MsgBox ("Number of trace elements returned = " & length)

'Create a string from the ResultsArray to output to a file
For x = 0 To length - 1
```

Programming Examples

```
      output = output & ResultsArray(x) & vbCrLf
    Next x

    'Print Results to the Screen
    MsgBox (output)

    'Store the results in a text file
    fnum = FreeFile()  'Get the next free file number
    Open "bintrace.txt" For Output As #fnum
    isOpen = True
    Print #fnum, output

    ' Intentionally flow into Error Handler to close file
Error_Handler:
    ' Raise the error (if any), but first close the file
    If isOpen Then Close #fnum
    If Err Then Err.Raise Err.Number, , Err.Description
    Exit Sub

VisaErrorHandler:
   Dim strVisaErr As String * 200
   Call viStatusDesc(defrm, status, strVisaErr)
   MsgBox "*** Error : " & strVisaErr, vbExclamation, "VISA Error Message"
   Exit Sub
End Sub
```

Using Visual Basic® .NET with the IVI-Com Driver

This example uses Visual Basic .NET with the IVI-Com driver. It makes a time domain (Waveform) measurement using the Basic mode. Basic mode is standard in the E4406A Vector Signal Analyzer and is optional (B7J) in the PSA Series Spectrum Analyzers. The vb file (vbivicomsa_basicwaveform.vb) and the compiled executable file (vbivicomsa.exe) can be found on the Documentation CD.

```
'**************************************************************************
' VBIviComSA_BasicWaveform.vb, August 5, 2003
' This example demonstrates the use of the IVI-COM driver in VB.NET
' through an interop assembly.  The Raw I/Q trace data from the Waveform
' measurement in Basic Mode is queried and printed to the screen.
'
' Requirements:
'  1) E4406A or PSA Series Spectrum Analyzer with Option B7J
'  2) Latest AgilentSa IVI-COM driver
'     You may download it here: http://www.agilent.com/find/inst_drivers
'     This example was tested with version 2.1.0.0 of the driver
'  3) Create a new project and add the References to this module
'     and to the the IVI-COM driver dlls:
'     For .NET, right click on Reference, choose Add Reference
'     and then click on Browse and directly link the DLLs in the directory:
'     C:\Program Files\IVI\Bin\Primary Interop Assemblies
'     Agilent.AgilentSa.Interop.dll
'     Agilent.AgilentSaAppBasic.Interop.dll
'     Agilent.Itl.Interop
'     IviDriverLib.dll
'     IviSpecAnLib.dll
'
' THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
' KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
' IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
' PARTICULAR PURPOSE.
'
```

Programming Examples

```
' Copyright (c) 2003. Agilent Technologies, Inc.
'*************************************************************************
Option Strict On

Imports Agilent.TMFramework
Imports Agilent.AgilentSa.Interop
Imports Ivi.Driver.Interop
Imports System.Runtime.InteropServices

Module ConsoleApp

  Sub Main()

    ' Prompt the user for the address of the instrument
    Dim address As String
    Console.WriteLine("Enter address of the instrument " & vbCrLf & _
    "(ex: GPIB0::18::INSTR or TCPIP0::192.168.100.2::inst0::INSTR):")
    address = Console.ReadLine()

    Try
      ' Create an instance of the driver, connection to the instrument
      ' is not established here, it is done by calling Initialize
      Dim instr As New AgilentSaClass()

      ' Establish the connection to the instrument
      ' Last parameter (DriverSetup) is optional, VB could omit it (but not C#)
      ' Important: Close must be called to release resources used by the driver
      instr.Initialize(address, False, False, "")

      Try
        ' INHERENT CAPABILITIES
        ' Note that it is not necessary to program against the IIviDriver
        ' interface, the same can be achieved by using the class directly
        ' Using the IIviDriver interface gives us interchangeable code
        Dim inherent As IIviDriver = instr
```

```
Dim manufacturer As String
Dim model As String
Dim firmware As String

manufacturer = inherent.Identity.InstrumentManufacturer
model = inherent.Identity.InstrumentModel
firmware = inherent.Identity.InstrumentFirmwareRevision
' Output instrument information to the console
Console.WriteLine("Manufacturer: " + manufacturer)
Console.WriteLine("Model: " + model)
Console.WriteLine("Firmware: " + firmware)

' Reset the instrument
inherent.Utility.Reset()

' INSTRUMENT SPECIFIC
' Using the IAgilentSa interface is not necessary or beneficial
' at the moment, but in the future if other instruments implement
' the IAgilentSa interface, the code that is written to work with
' that interface can be reused without changes, as opposed to code
' that is written against the class object directly
Dim sa As IAgilentSa = instr

' Obtain trace data from the instrument
Dim traceData As Array
sa.Application.Select("Basic")
'sa.Application.Basic.Waveform.Configure()
sa.Application.Basic.Spectrum.Traces.Initiate()
traceData = sa.Application.Basic.Waveform.Traces.Item("RawIQ").Read(10000)

' Output the trace data to the console
Console.WriteLine("Press ENTER to display trace data.")
Console.ReadLine()
Dim traceValue As Double
For Each traceValue In traceData
  Console.WriteLine(traceValue)
```

```
            Next

          Catch ex As Exception
            Console.WriteLine(ex.Message)

          Finally
            ' Close the connection
            instr.Close()

          End Try

        Catch ex As COMException
          Console.WriteLine(ex.Message)

        Catch ex As Exception
          Console.WriteLine(ex.Message)

        End Try

        ' Wait for user input
        Console.WriteLine("Press ENTER to end program.")
        Console.ReadLine()

      End Sub

    End Module
```

Index