

4 DEBUGGING

In This Chapter

This chapter contains the following topics:

- “Debug Sessions” on page 4-2
- “Code Behavior Analysis” on page 4-9
- “Program Execution Operations” on page 4-12
- “Simulation Tools” on page 4-16
- “Plots” on page 4-17

Debug Sessions

You run the DSP projects that you develop as *sessions* (debug sessions).

A session is defined by the elements described in the following table.

Element	Description
Debug target	The debug target is the software module that controls a type of debug target (a simulator or emulator). The simulator is software that mimics the behavior of a DSP chip. Simulators are used to test and debug DSP code before a DSP chip is manufactured. An emulator is software that “talks” to a hardware board that contains one or more actual DSP chips.
Platform	For a given debug target, several platforms may exist. For a simulator, the platform defaults to the identically named DSP simulator. When the debug target is an EZ-ICE board, the platform is the board in the system on which you want to focus. When the debug target is a JTAG emulator, the platforms are the individual JTAG chains.
Processor	Multiple processors (emulation only) can exist for a given debug target and platform. When you create an executable file, the processor is specified by the Linker Description File (.LDF) and other source files.

When you set up a session, you set the focus on a series of increasingly more specific elements.

The target platform and processor settings specify the debug session. A default session name is automatically generated. You can further identify the session by modifying the default name, choosing a more meaningful name. Note that a well-chosen name can prevent confusion later.

Debug Session Management

You can set up several debug sessions at once and dynamically switch between them.

You typically run multiple debug sessions to write different versions of your program to compare their operating efficiencies. Another reason for running multiple sessions is to debug completely different programs without having to run multiple instances of VisualDSP++.

Simulation vs. Emulation

Connecting to a simulator session enables you to open as many sessions as your system's memory can handle.

Connecting to actual hardware through an emulator enables you to have only **one** debug session connected to one emulator at any time. If you install multiple emulators and connect them to multiple target boards, you can connect only one debug session to each individual emulator.

Note: Connecting to a JTAG emulator enables you to have only **one** debug session connected to each physical target/emulator combination. Otherwise, contention issues may arise. If you switch to a different session, VisualDSP++ detaches from the old session before attaching to the new session.

MP Debug Sessions vs. Single-Processor Debug Sessions

Often, performance-based products require two or more DSPs. A system built with multiple DSPs is called a *multiprocessor* system, and a system built with a single DSP is called a *single-processor* system.

Note: You can perform multiprocessor debugging in emulator sessions only. This version of the TigerSHARC simulator does not support multiprocessor debugging.

Multiprocessor (MP) commands work like single-processor commands, except that they work synchronously on **all active processors** in the currently selected MP group. To debug individual processors in an MP session, use pinning and the processor status items in the Multiprocessor window with single-processor debug commands. See [“Focus and Pinning Features” on page 4-7](#).

You can set breakpoints in your executable program. You can set a breakpoint at any address in program memory. Program execution halts at the address at which the breakpoint is located.

Note: In addition to software breakpoints, you can also use hardware breakpoints in an emulator debug session.

Multiprocessor (MP) Debug Session

A *multiprocessor* system consists of multiple DSPs. In a multiprocessor debug session (emulation only), you synchronously run, step, halt, and observe program execution operations in all the processors at once.

The following capabilities help to speed a multiprocessor debug session:

- Multiprocessor debug commands that operate like the single-processor debug commands
- Multiprocessor window
 - The **Status** tab enables you to view the status of each processor and switch processor focus (see [“Focus and Pinning Features” on page 4-7](#))
 - The **Group** tab enables you to group processors into multiple, logical units to which all MP commands are applied
- Window pinning (see [“Focus and Pinning Features” on page 4-7](#))
- Window color specification (see [“Additional Focus Indication” on page 4-8](#))

Debug Sessions

Setup

The first step in setting up a multiprocessor debug session is to develop a multiprocessing project by using the multiprocessing capabilities of the linker and an `.LDF` file to describe the multiprocessing system.

Refer to your DSP's Linker and Utilities Manual, especially the sections about `SHARED_MEMORY{ }` and `MPMEMORY{ }` commands.

The second step is to use the JTAG-ICE Configurator utility to describe the JTAG emulator hardware to the VisualDSP++ software. VisualDSP++ uses this description when you set up your debug session. Refer to your DSP's Hardware Specification for information about the JTAG-ICE Configurator. After specifying your hardware system, build your project.

The first time that you launch VisualDSP++ for a new project, the **New Session** dialog box opens to enable you to configure the MP session. The next time that you launch VisualDSP++, the debug session is automatically configured for you.

Focus and Pinning Features

In a multiprocessor debug session, you often have to examine the behavior of a single processor to better understand its interaction with the other processors on the target.

When you debug a single processor in an MP session, the processor being debugged has the *focus*.

By *pinning a window* to a processor, you dedicate the window, such as a Memory window, to a particular processor in a multiprocessor group. Pinning statically associates a window to a specific processor.

Tip: Before debugging, open and pin the register windows and Memory windows you plan to use. If you do not pin them, these windows display information for any processor that has focus.

When a window is pinned to a processor, a pin icon  appears in the window's upper-left corner.

For example: 

Debug Sessions

Window Title Bar Information

Figure 4-1 shows a pinned window in a multiprocessor debug session.

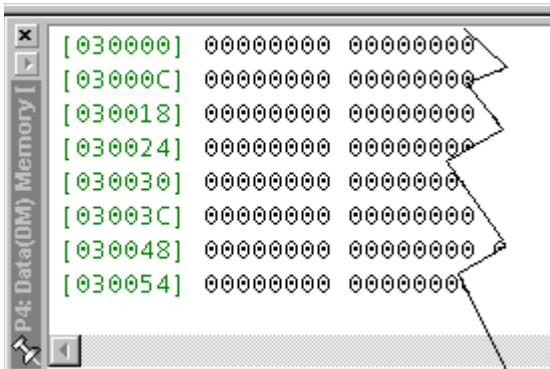


Figure 4-1. Pinned Window in a Multiprocessor Debug Session

The title bar of a pinned window shows:

- Processor name
- Pushpin icon  to indicate that the window is pinned
- Window title
- Number format, such as Hexadecimal (for windows that support multiple formats)

Additional Focus Indication

If configured, VisualDSP++ shades unfocused windows with a specified color. You can specify the background color of focused and unfocused windows.

Code Behavior Analysis

Code behavior analysis involves examining and analyzing code execution to locate areas that may be optimized for better performance.

VisualDSP++ provides the following tools for code behavior analysis:

- Statistical profiles and Linear profiles

These two profiling methods measure program performance by sampling the target's PC register to collect data.

Use both methods during program development. Simulator targets support linear profiling. JTAG emulator targets support statistical profiling.

- DSP memory plots

Plots enable you to configure data sets, process the data, and then display a representation of DSP memory

Statistical Profiling vs. Linear Profiling

VisualDSP++ provides two profiling methods that measure program performance by sampling the target's PC register to collect data.

Simulator targets support linear profiling. Emulator targets support statistical profiling.

The Linear Profiling Results window and Statistical Profiling Results window display the data collected by these two profiling methods and indicate where the application is spending its time.

The window's title (**Linear Profiling Results** or **Statistical Profiling Results**) depends on whether this tool is used during simulation or emulation.

Code Behavior Analysis

Simulation

Linear profiling with the simulator is not statistical because the simulator samples **every** PC executed. This feature provides an accurate and complete picture of what was executed in your program.

Linear profiling is much slower than statistical profiling. Simulator targets support linear profiling but do not support statistical profiling.

Emulation

A statistical profile measures the performance of a DSP program by sampling the target's Program Counter (PC) register at random intervals while the target is running the DSP program. The areas of the program where most of the PCs are concentrated are where most of the time is spent in executing the program.

Statistical profiling provides a more generalized form of profiling that is well suited to JTAG emulator debug targets. Emulator targets do not support linear profiling.

JTAG sampling is completely non-intrusive, so the process does not incur additional run-time overhead.

DSP Memory Plots

You can display DSP memory as a *plot* in a Plot window, as shown in Figure 4-2.

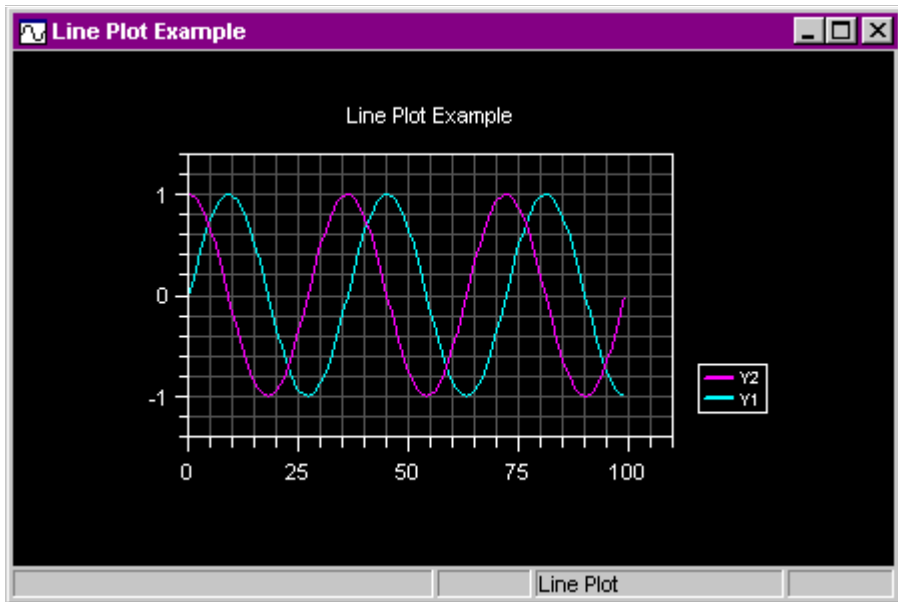


Figure 4-2. Plot Window – Display of DSP Memory

You can visualize the DSP memory data and process it by using a data processing algorithm. You can choose from multiple plot types and can specify the plot's data and presentation.

You can modify a plot's configuration and immediately view the revised plot. From a Plot window, you can zoom in on a portion of a plot or view the values of a data point. You can print a plot, save the plot image to a file, or save the plot's data to a file.

Program Execution Operations

Program execution operations include loading the executable program, using program execution commands, restarting the program, and using breakpoints.

Loading the Executable Program

After completely specifying the debug session, the last step before you can begin the debug session is to load the DSP executable program. If you launch VisualDSP++ in stand-alone mode, ensure that the session is configured correctly **before** you load your program.

After a successful build of the target executable, VisualDSP++, if configured, loads the executable automatically to the current session when the session processor type matches project's processor. When the current session processor does not match the project's processor type, you are prompted to choose another session.

If automatic load is not configured, VisualDSP++ does not try to load the executable automatically after a successful build. Note that the target must be an executable (.EXE) file.

This debugging feature saves time, as you do not have to load the executable target manually, and you can start to debug right after a successful build of the project.

Using Program Execution Commands

You can run program execution commands from the **Debug** menu or by clicking toolbar buttons. Executable files run until an event such as a breakpoint or user-issued **Halt** command stops execution. When program execution halts, all windows are updated to current addresses and values.

Multiprocessor (MP) commands (emulation only) operate like single-processor commands with one exception—they perform an action on **all** processors in the MP group. **MP Run**, **MP Halt**, and **MP Step** are synchronous operations, which means that all processors in the currently selected group execute on exactly the same clock cycle. Some commands have keyboard shortcuts.

Use the following commands to control program execution:

Command	Description
Run and MP Run	Runs an executable. The program runs until an event stops it, such as a breakpoint or user intervention. When program execution halts, all windows update to current addresses and values.
Halt and MP Halt	Stops program execution. All windows are updated after the DSP halts. Register values that have changed are highlighted, and the status bar displays the address where the program halted.
Run to Cursor	Runs the program to the line where you left your cursor. You can place the cursor in Editor windows and Disassembly windows.
Step Over	(C/C++ code only in an Editor window) Single-steps forward through program instructions. If the source line calls a function, the function executes completely, without stepping through the function instructions.
Step Into and MP Step Into	(Editor window or Disassembly window) Single-steps through the program one C/C++ or assembly instruction at a time. Encountered functions are entered.
Step Out Of	(C/C++ code only in an Editor window) Performs multiple steps until the current function returns to its caller, and stops at the instruction immediately following the call to the function.

Restarting the Program

You can set the program counter (PC) to the first address of the interrupt vector table.

Performing a Restart during Simulation

In the simulator, restart works like a reset; however, the target's memory does not change. All registers are reset to their initial values.

Note: Memory is not reset. Thus, C and assembly global variables are **not** reset to their original values. Your program may behave differently after a restart. To re-initialize these values, reload your `.DXE` file.

Performing a Restart during Emulation

In the emulator, restart works exactly like a reset. Only registers with default reset values are affected. All other registers remain unchanged.

Using Breakpoints

An enabled breakpoint halts program execution at a specific instruction or address. You can enable and disable breakpoints as well as add and delete breakpoints.

A disabled breakpoint is set up, but not turned on. A disabled breakpoint does not stop program execution. It is dormant and may be used later.

A *break* occurs when the conditions that you specify are met.

Symbols in the left margin of a Disassembly window or Editor window indicate breakpoint status.

Symbol	Indicates
	An enabled (set) breakpoint
	A disabled breakpoint (recognized, but cleared)

Unconditional Breakpoints

You can configure a breakpoint to occur when the program counter (PC) reaches a specific address. This type of breakpoint is called an *unconditional breakpoint* because it occurs when the address is reached.

Simulation Tools

Note: You can quickly place an unconditional breakpoint at an address in a Disassembly window or Editor window by using one of these options:

- Select the address and click the **Toggle Breakpoint** button .
- Double-click on the line in the Disassembly or Editor window.

Conditional Breakpoints

You can configure a breakpoint to occur when various conditions (criteria) are met. This type of breakpoint is called a *conditional breakpoint*. The conditions may include:

- A user-defined *expression* that must evaluate to `TRUE`
- A *skip* (count) that specifies the number of times to skip over the breakpoint before finally halting

If both an expression and skip are set, execution stops when the breakpoint is reached “*n*” times when the expression is `TRUE`, where *n* represents the skip count. When the expression is empty, execution stops when the breakpoint is reached “*n*” times.

Simulation Tools

You can simulate the core processor to develop, test, and debug algorithms. In addition, you can simulate external interrupts. The simulator simulates the memory specified in the architecture file. VisualDSP++ lets you interactively observe and alter the data in the processor and in memory.

Plots

Plots enable you to configure data sets, process the data, and display representations of values obtained from DSP memory. The plot presentation options (types) are described below.

Plot Types

You specify a plot as one of these plot types:

Plot Type	Description	Requires
Line plot	Displays points connected by a line	Y value for each point
X-Y plot	Similar to a line plot, but also uses X-axis data	X value and Y value for each data point
Constellation plot	Displays a symbol at each data point	X value and Y value for each data point
Eye diagram	Typically used to show the stability of a time-based signal	Y value for each data point
Waterfall	3-D plot typically used to show the change in frequency content of signal over time	Z value for each data point
Spectrogram plot	2-D plot displays amplitude data as a color intensity	Z value for each data point

Plots

The X, Y, and Z values are read from DSP memory.

Line Plots

A line plot (shown in [Figure 4-3](#)) displays a range of DSP memory values connected by a line. The values read from DSP memory are assigned to the Y-axis. The corresponding X-axis values are automatically generated.

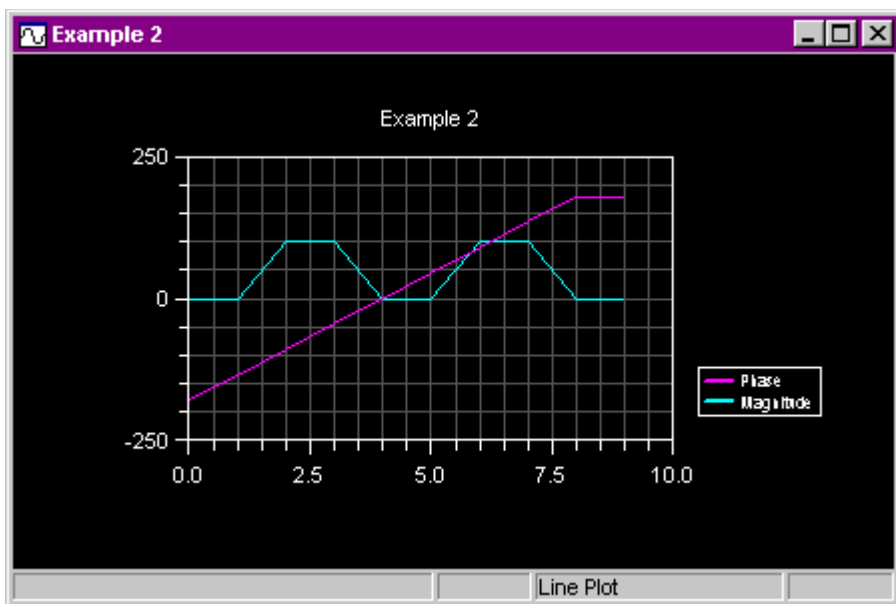


Figure 4-3. Line Plot Example

You can plot multiple data sets on a single graph.

X-Y Plots

An X-Y plot (shown in [Figure 4-4](#)) requires an X value and a Y value for each data point. Unlike a line plot, an X-Y plot requires that you specify the X-axis data.

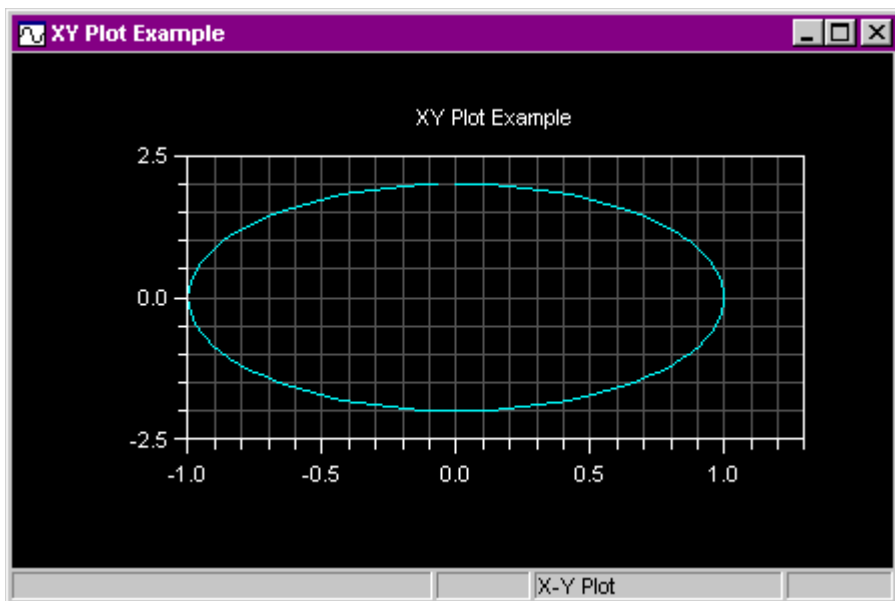


Figure 4-4. X-Y Plot Example

The X data and Y data are specified separately in a user-defined memory location. The number of X and Y points must be equal.

Constellation Plots

A constellation plot (shown in [Figure 4-5](#)) displays a symbol at each (X,Y) data point.

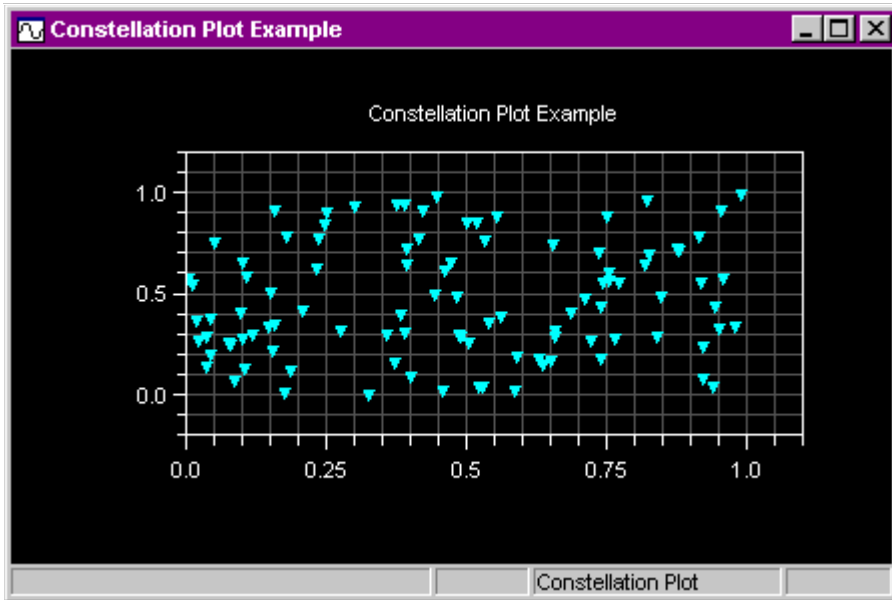


Figure 4-5. Constellation Plot Example

The X and Y data are specified separately in a user-defined DSP memory location. The number of X and Y points must be equal.

Eye Diagrams

An eye diagram plot (shown in [Figure 4-6](#)) is typically used to show the stability of a time-based signal. The more defined the eye shape, the more stable the signal.

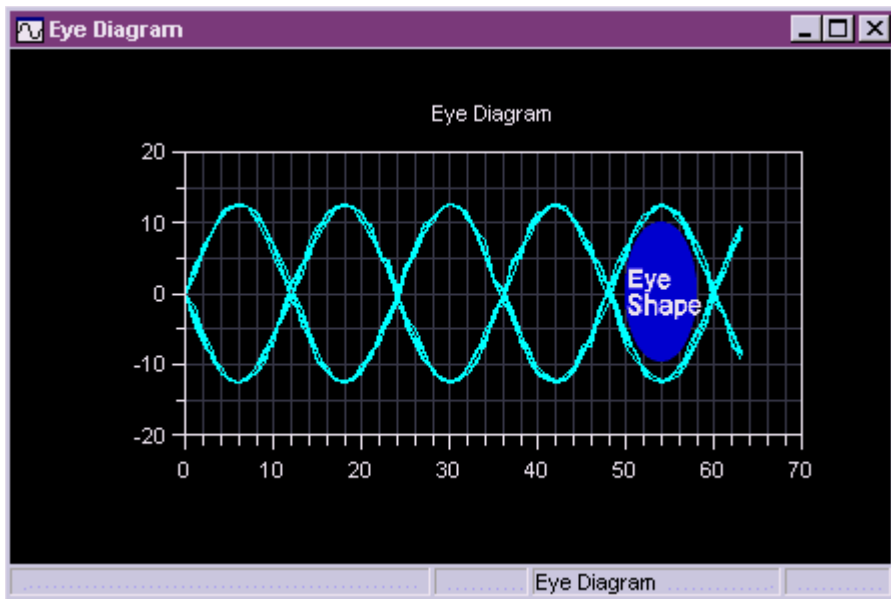


Figure 4-6. Eye Diagram Plot Example

This plot works like a storage oscilloscope, displaying an overlapped history of a time signal. The eye diagram plot processes the input data and optionally looks for a threshold crossing point (default 0.0). The trace is plotted when the threshold crossing is reached, and it continues plotting for the remainder of the trace data.

Plots

When a breakpoint occurs (or a step is performed), the plot data is updated and a new trace is plotted. The eye diagram uses a data shifting technique that stores the desired number of traces in a plot buffer (default is ten traces). Upon exceeding the number of traces, the first trace shifts out of the buffer and the new trace shifts into the last buffer location. This technique is referred to as first-in, first-out (FIFO).

You can modify options for threshold value, rising trigger, falling trigger, and the number of overlapping traces.

Waterfall Plots

A waterfall plot (shown in [Figure 4-7](#)) is typically used to show the change in frequency content of signal over time.

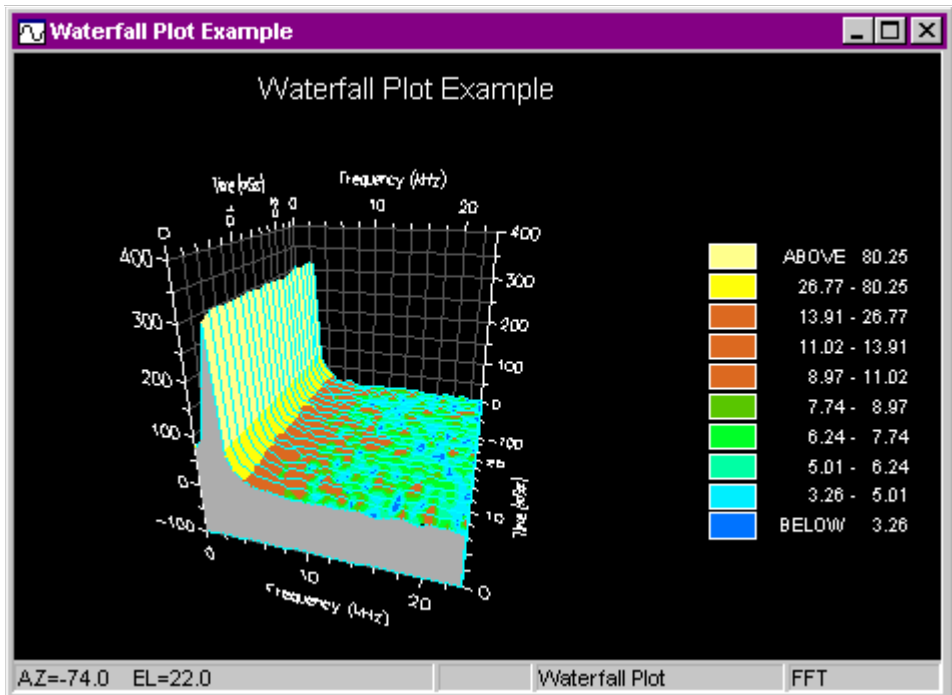


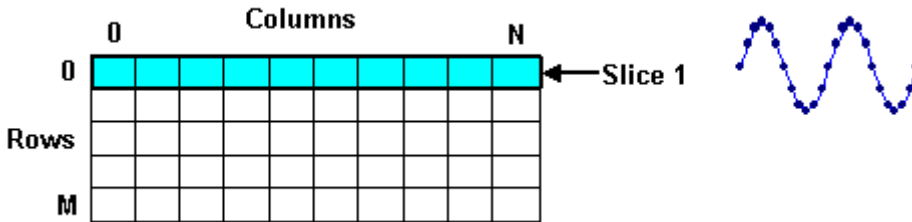
Figure 4-7. Waterfall Plot Example

The plot comprises multiple line plot traces in a three-dimensional view. Each line plot trace represents a slice of the waterfall plot.

The easiest way to create a waterfall plot is to define a two-dimensional array (a *grid*) in C code. The first array dimension is the number of rows in the grid, and the second dimension is the number of columns in the grid. The number of columns is equal to the number of data points in each line trace.

Plots

A time-based signal is sampled at a predefined sampling rate and stored as a slice in the grid (row 0, columns 0 through N).



The next time signal is sampled and stored (in row 1, columns 0 through N). This process continues until all the rows are filled.

By default, an FFT performed on each slice results in a frequency output display. You can enhance the display by using a color map (specified on the **3-D Axis** tab page of the **Color Settings** dialog box). Each color corresponds to a range of amplitude values.

The plot output displays a legend that shows each color and associated range of values.

You can rotate the waterfall plot to any desired azimuth and elevation by using the keyboard's arrow keys.

Spectrogram Plots

A spectrogram plot (shown in [Figure 4-8](#)) displays the same data as a 3-D waterfall plot, except in a 2-dimensional format.

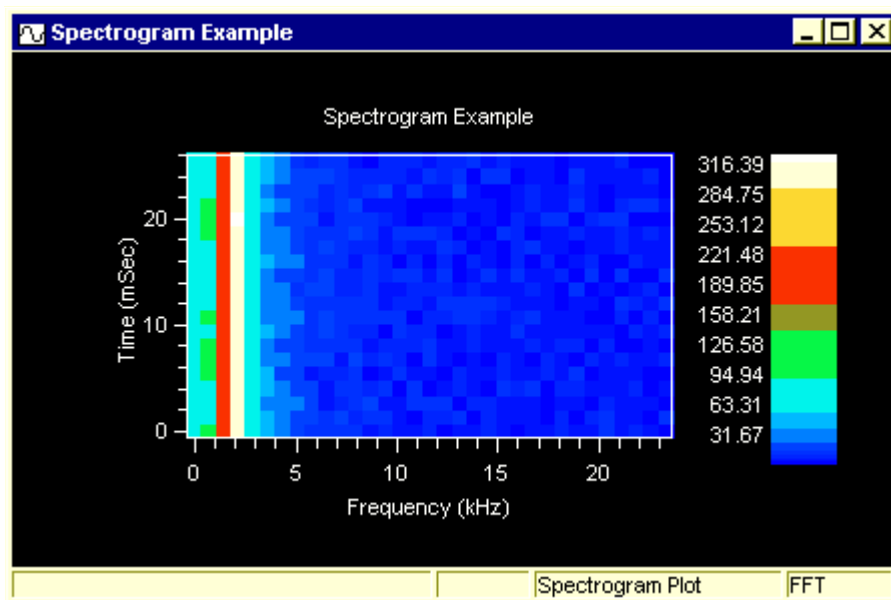


Figure 4-8. Spectrogram Plot Example

Each (X,Y) location displays a color representing the amplitude of the data. By default, an FFT is performed on each slice, which results in a frequency output display. A legend displays the colors and associated range of values.

Plots