User’s Guide

Publication number B4605-97006
October 2002

For warranty information, see the pages behind the index.

Tool Development Kit

Tool Development Kit at a Glance

The Tool Development Kit is a programming environment that allows access to data acquired by Agilent Technologies 16600 or 16700 series logic analyzers. This data can be retrieved and manipulated through the use of a Tool Development Kit program. A library of functions and type definitions that are specifically for use with the Tool Development Kit is provided.

Tool Development Kit programs are written in the C programming language, with some C++ extensions, and must be compiled within the Tool Development Kit tool. The Tool Development Kit text editor provides basic text editor capabilities using the mouse and keyboard. Program files can be written within the Tool Development Kit or they can be imported.

This manual is divided into eight chapters.

- Chapter 1, “Getting Started,” provides instructions for installing and enabling the Tool Development Kit.
- Chapter 2, “Welcome to Tool Development Kit,” provides instruction in a tutorial format for the basic start-up tasks required to use the Tool Development Kit.
- Chapter 3, “Tool Development Kit Interface,” provides information on using the Tool Development Kit interface.
- Chapter 4, “Tool Development Kit Concepts,” covers concepts fundamental to using the Tool Development Kit tool to write programs that manipulate data of interest.
- Chapter 5, “Tool Development Kit Programming Model,” covers the best practices for using Tool Development Kit and writing programs.
- Chapter 6, “Tool Development Kit System Utilities,” covers the Tool Development Kit system utility functions.
- Chapter 7, “Extended Examples,” covers more complex examples to demonstrate real-world functionality.
- Chapter 8, “Tool Development Kit Reference,” covers Tool Development Kit functions in alphabetical order.

Contents

Contents

8 Tool Development Kit Reference

1

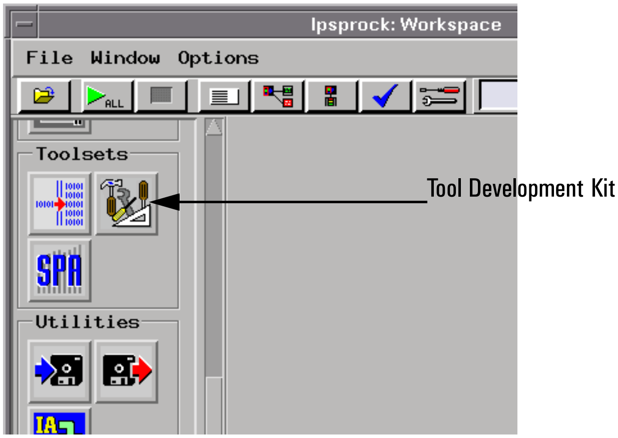
Getting Started

Verifying Tool Development Kit Installation

Before you get started using the Tool Development Kit, make sure that it is installed and ready for use. Analyzers ordered with the Tool Development Kit should have it ready to use.

1 Select the **Workspace** button from the toolbar.

2 In the Workspace window, look for the Tool Development Kit icon in the Toolsets group of icons.

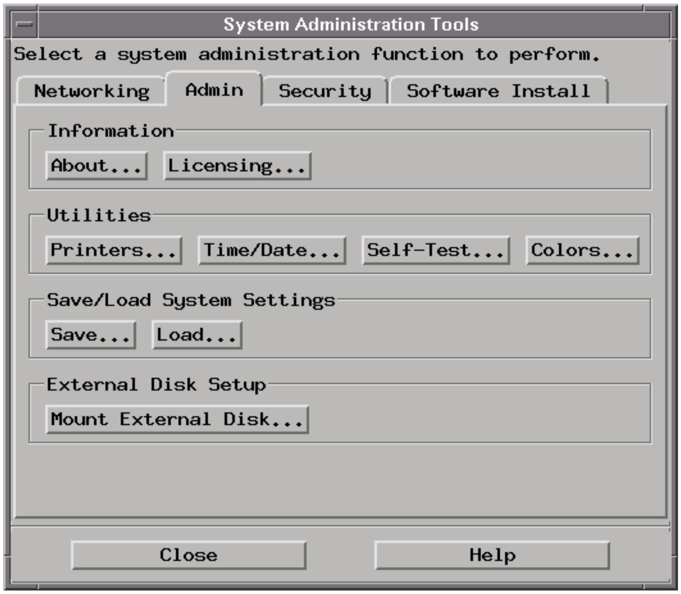


3 If the Tool Development Kit icon is there, go to chapter 2 to begin the tutorial. If the icon is not there, go to the next step.

4 Select the **System Admin** button from the toolbar.

5 Select the **Admin** tab.

6 Select the **Licensing...** button.



Select a system administration function to perform.

7 Look for the Tool Development Kit listed under “Product.”

- If it is listed, go to “Licensing the Tool Development Kit” on page 18.
- If it is not there, go to “Installing the Tool Development Kit” on page 16.

Installing the Tool Development Kit

If the CD-ROM drive is not connected, see the instructions printed on the CD-ROM package.

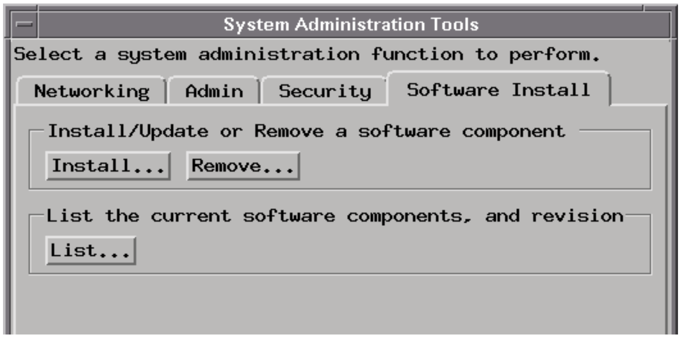
1 Turn on the CD-ROM drive first and then turn on the logic analysis system.

2 Insert the CD-ROM in the drive.

3 Select the **System Admin** button from the toolbar.

4 Select the **Software Install** tab.

5 Select the **Install...** button.



System Administration Tools

Select a system administration function to perform.

Networking
Admin
Security
Software Install
Install/Update or Remove a software component
Install...
Remove...
List the current software components, and revision
List...

Change the media type to “CD-ROM” if necessary.

6 Select the **Apply** button.

7 Open the “AUXILIARY-SW” selection.

8 Select the Tool Development Software, then select **Install...**.

The dialog box will display “Progress: completed successfully” when the installation is complete.

9 Select **Close** to close the Software Install window.

10 Select **Close** to close the System Administration Tool window.

11 Go to page 18 and complete the steps for licensing the Tool

Development Kit.

See Also The instructions printed on the CD-ROM package for a summary of the installation instructions.

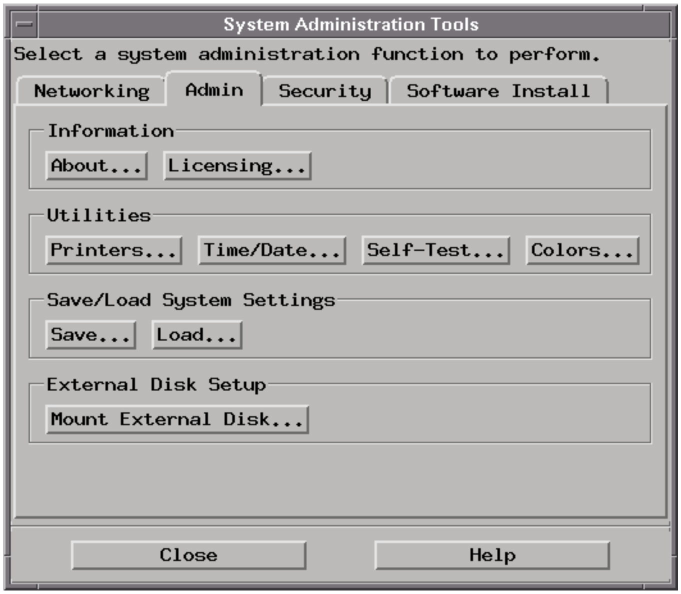
Licensing the Tool Development Kit

1 To obtain a password, contact the Agilent Technologies password center listed on the Entitlement Certificate that you received after you purchased the Tool Development Kit.

2 Select the **System Admin** button from the toolbar.

3 Select the **Admin** tab.

4 Select the **Licensing...** button.



Select a system administration function to perform.

5 Enter the Password into the field labeled “Tool Development Kit”.

6 Select **OK.**

7 Select **Close** to close the System Administration Tools window.

Now the Tool Development Icon is visible on the workspace

8 You may now proceed to Chapter 2 and begin using the Tool Development Kit tool.

Response Center Support

One year of response center support for two people is included with this product. You will receive a letter in a few weeks with your contract and contact information.

For further information about this support:

1. Go to:

   http://www.agilent.com/find/swtools

2. Select the link for the Tool Development Kit page, Agilent Technologies B4605B.
3. Select Agilent Technologies B4605B Technical Support, which is located on the right hand side of the page.
4. Under “Professional Services”, select Services.
5. Under “Technical and Professional Services”, select Software Support and Services.
6. Under “Technical and Professional Services”, select Response Center Support.

2

Welcome to Tool Development Kit

Tool Development Kit Overview

The Tool Development Kit is a programming environment that allows access to data acquired by Agilent Technologies 16600 or 16700 series logic analyzers. This data can be retrieved and manipulated through the use of a Tool Development Kit program. A library of functions and type definitions that are specifically for use with the Tool Development Kit is provided.

Tool Development Kit programs are written in the C programming language, with some C++ extensions, and must be compiled within the Tool Development Kit tool. The Tool Development Kit text editor provides basic text editor capabilities using the mouse and keyboard. Program files can be written within the Tool Development Kit or they can be imported.

Central to using the Tool Development Kit to retrieve and manipulate data is an understanding of the different types of data that can be captured by the logic analyzer. This will be discussed in Chapter 4. It is recommended that you read this material before creating your own Tool Development Kit programs. In order to use the Tool Development Kit, you must either set up a workspace or open an existing configuration. The rest of this chapter walks you through the logistics of setting up a workspace, opening a configuration, the Tool Development Kit tool and a program file, and then finally creating your first program with the Tool Development Kit editor.

The sample files, which are all in the directory /logic/demo/ToolDevKit/, are provided in “read-only” mode. If you wish to modify them, they can be copied and changed as you wish.

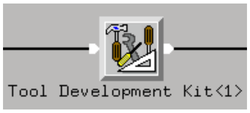
Setting Up the Workspace

Before writing a Tool Development Kit program it is necessary to set up the workspace. A data source, the Tool Development Kit tool, and one or two listers comprise a typical configuration. The data source can be either a Logic Analyzer instrument or a File In tool.

For the following examples, a File In tool that loads a previously acquired trace into the data flow is used as the data source. This example shows two ways of establishing a data flow: dropping one tool on top of another, and dragging from one tool's output to another tool's input.

1 Drag a File In tool to the workspace.

2 Drag a Tool Development Kit tool onto the workspace.



Tool Development Kit tool showing data input and output points

3 Connect data inputs and outputs to establish data flow.

  a Drag from the output point of the File In tool to the input point of the Tool Development Kit tool.

4 Drag a Lister to the workspace and drop it on top of the Tool Development Kit tool.

5 If desired, drag and drop another Lister on top of the File In tool. This is useful if you wish to view the data without it being processed by the Tool Development Kit tool.

6 Open a data file in the File In tool.

  a Select the File In tool and choose Display.

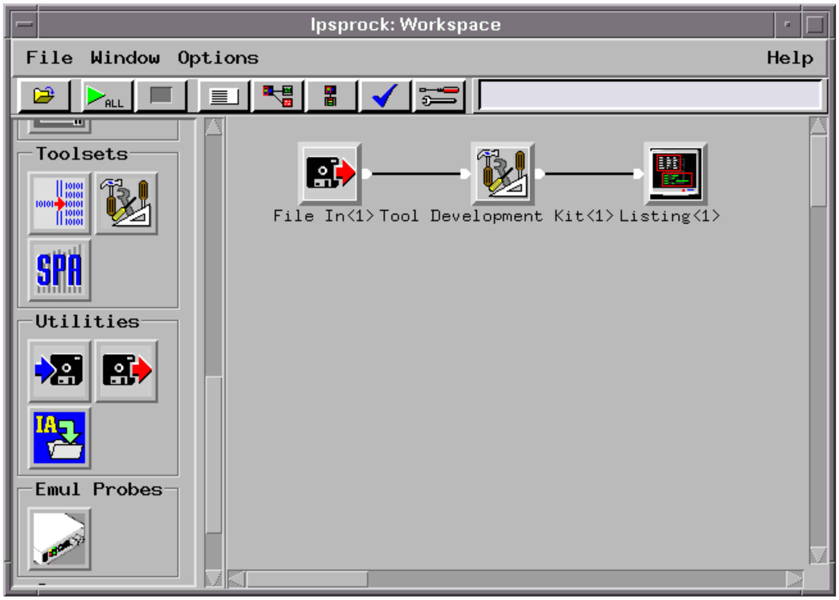
  b Enter /logic/demo/ToolDevKit/sample1.dat for file name.

c Select the Read File button, then Close

7 Save this configuration.

a Choose File → Save Configuration from the main window menu bar.

b Give the file a name of your choosing.



Workspace window showing placement of tools and data flow

These steps create a workspace consisting of a data source, the Tool Development Kit tool, and displays so that data from the source flows “down the wire” to the Tool Development Kit tool and displays.

Opening a Configuration, the Tool Development Kit Tool and a Program File

For each Tool Development Kit session, you may set up a workspace as described in the previous section, or open a previously saved configuration. It is also necessary to open the Tool Development Kit tool in preparation for program code entry or opening an existing program file.

This example demonstrates opening a previously saved configuration, opening the Tool Development Kit tool, and loading an existing program file.

1 Open the configuration

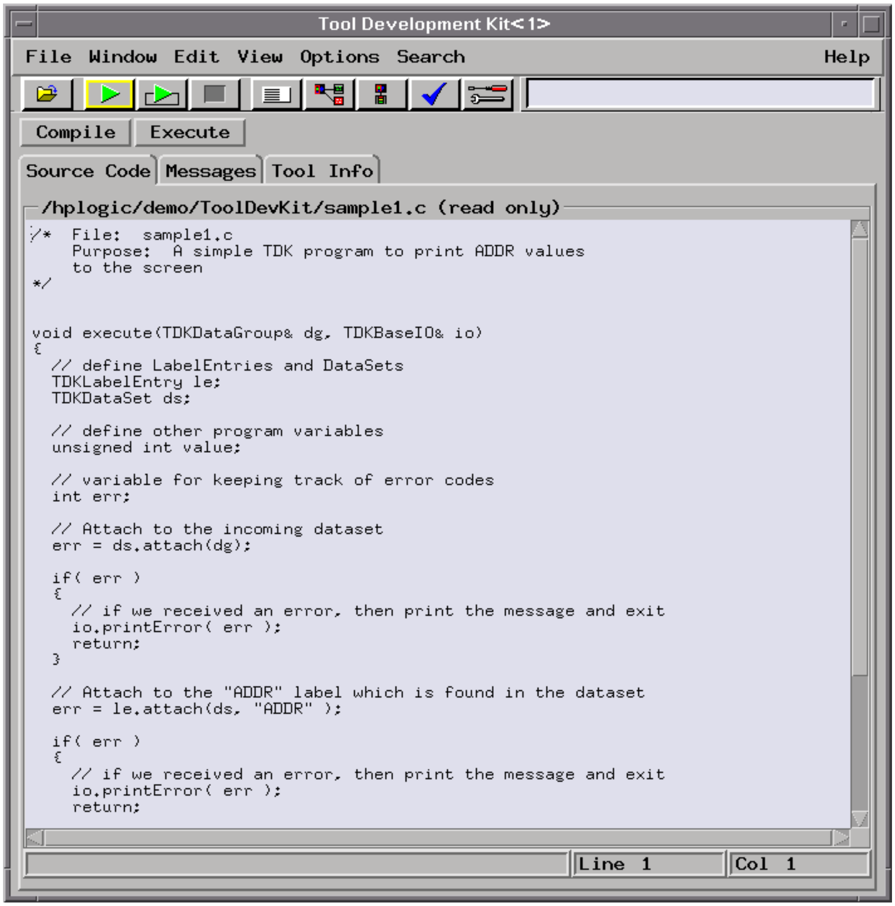
- **a** Choose File ➔ Load Configuration from the Workspace window menu bar.
- **b** Choose /logic/demo/ToolDevKit/sample1.___ as the config name.

2 Open the Tool Development Kit tool.

- **a** Select the Tool Development Kit tool and choose Display

3 Open a Tool Development Kit program file

- **a** Choose File ➔ Open Source file from the Tool Development Kit menu bar
- **b** Choose /logic/demo/ToolDevKit/sample1.c



/hplogic/demo/ToolDevKit/sample1.c (read only)

```
/*  File:  sample1.c
    Purpose:  A simple TDK program to print ADDR values
    to the screen
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  // define LabelEntries and DataSets
  TDKLabelEntry le;
  TDKDataSet ds;

  // define other program variables
  unsigned int value;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the "ADDR" label which is found in the dataset
  err = le.attach(ds, "ADDR" );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
```

The open Tool Development Kit tool, editor and sample1.c program

These primary steps will be used in all of the following examples. When the Tool Development Kit tool has been opened, the Tool Development Kit editor is available to enter program code.

Creating Your Tool Development Kit Program - The Basics

Creating a Tool Development Kit program involves setting up or opening a configuration and using the Tool Development Kit editor to enter the C source code.

For this example, you will use the editor to enter the program. All the remaining sample programs will be automatically opened in the Tool Development Kit when you open the appropriate configuration.

NOTE: You do not necessarily have to perform the actual entry of code in this or any of the examples. All the code is provided on line in the **`/logic/demo/ToolDevKit`** directory.

1. Open the Tool Development Kit tool.
2. Enter the program sample1.c shown on the following pages using the Tool Development Kit editor.
3. Save the program file.
   - **a** Choose File ➔ Save Source File As... from the Tool Development Kit menu bar.
   - **b** Name the file /logic/demo/ToolDevKit/my_sample.c.
4. Select the Compile button.
5. Drag a File In tool onto the workspace.
   - **a** Open the data file /logic/demo/ToolDevKit/sample1.dat in the File In tool.
   - **b** Select the Read File button, then Close.
   - **c** Connect data inputs and outputs to establish data flow from the File In tool to the Tool Development Kit tool.

6 Notice that as soon as the data output of the File In tool is connected to the data input of the Tool Development Kit tool, a "Run" is performed.

The end of this section contains a listing of the source code sample1.c. It will be useful to reference it for this next discussion.

This sample program demonstrates several important concepts of the Tool Development Kit tool. The programming language used for the Tool Development Kit is standard ANSI C, with some C++ extensions. Use of C++ programming features beyond what is described here is not recommended. The Tool Development Kit library contains function calls and type definitions specifically for use with the Tool Development Kit.

The "main" function of C is replaced with the "execute" function call and should be used with the parameters shown in the sample program. The first parameter *dg* passed into the "execute" function is a reference to a variable of type TDKDataGroup. All programs will use this variable to access the data captured and passed in the tool. A data group contains one or more data sets. A data set contains one or more label entries. See Chapter 4, Tool Development Kit Concepts, for more information on data groups, data sets, and label entries.

In order to access and iterate through data, it is necessary to "attach" a variable name to the data set (TDKDataSet) and to the appropriate label entry (TDKLabelEntry). This can be accomplished by declaring two local variables. In this example, those variables are *ds* and *le* of type TDKDataSet and TDKLabelEntry. In a typical program, at least one data set will be attached to a data set contained within the incoming data group *dg*. In this sample, each attach function is error checked.

Notice that when a label entry is attached to a data set, a name is given to which that label entry is to be attached. In this example, *le* is attached to the label entry called "ADDR". In this example, neither a new data set nor label entry have been created. Rather local variables are used to access the incoming data set with label entry "ADDR".

Tip: an easy way to view the names of the data set and label entries contained within the data group passed into the Tool Development Kit program is to use the View->datagroup... option found on the Tool Development Kit menu bar. This dialog presents some very useful information that will be needed in order to perform certain tasks in a Tool Development Kit program. For this specific example, there is one data set contained in the data group. That data set is named "dataSet001" and contains 512 samples. Expand the data set folder to view the label entries contained in the data set. This data set contains three label entries, "ADDR", "DATA", and "STAT". For each label entry, a width in bits in shown together with the type of data.

The second parameter, *io*, passed into the "execute" function provides for displaying output messages to the Output window of the Tool Development Kit tool. Display of output messages is accomplished by using the io.printf() function with the same formatting as the C printf() statement. In addition to printing messages for user information, this is a method for debugging or monitoring values of variables. It should be noted that Tool Development Kit programs containing functions in addition to the required "execute" function will require the *io* parameter to be passed in if output messages are to be displayed from within that function. See the section on I/O System for more information on using *io* print functions. Similarly, if additional functions require access to the data group parameter *dg*, it must also be passed into that function in the same manner as the *io* parameter.

To summarize, a basic Tool Development Kit program will contain the following:

- the function "execute" with parameters *dg* (TDKDataGroup&) and *io* (TDKBaseIO&)
- a data set variable which will be attached to the incoming data group
- an integer variable *err* for storing the results of Tool Development Kit library function calls

Most likely, a Tool Development Kit program will also contain:

- one or more label entry variables which will be attached to one or more labels in the data set

Files Used:

From the /logic/demo/ToolDevKit/ directory.

- sample1.c (Tool Development Kit program file)

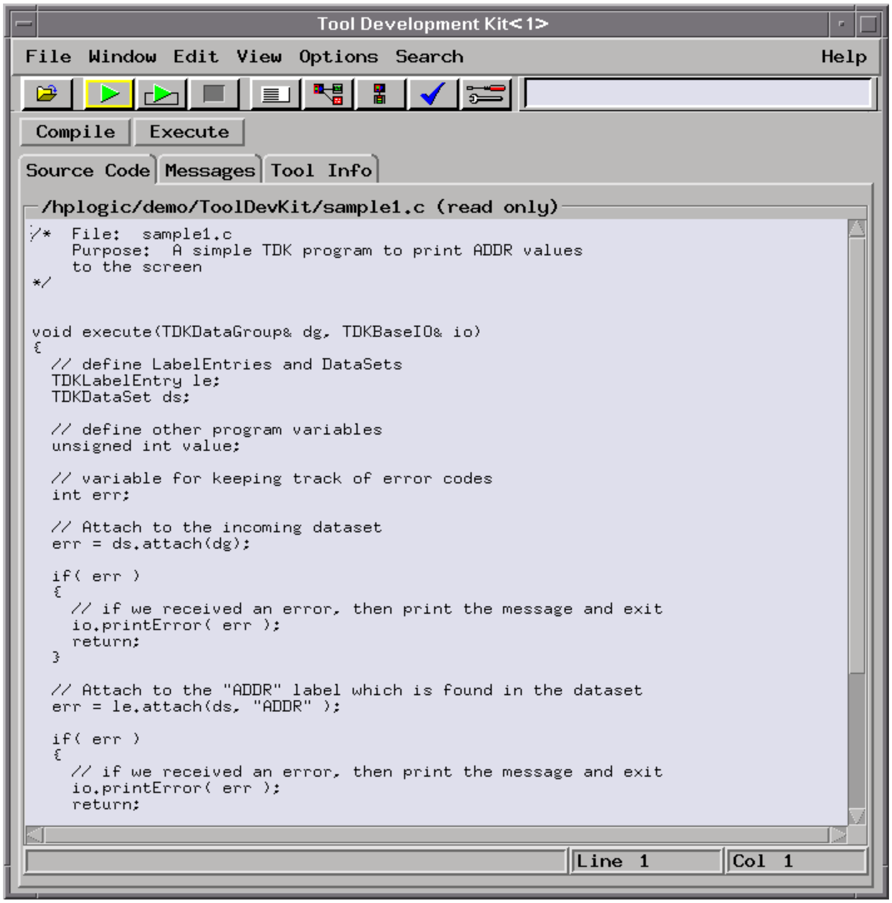
sample1.c

```
/*  File:  sample1.c
    Purpose:  A simple TDK program to print ADDR values
    to the screen
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  // define LabelEntries and DataSets
  TDKLabelEntry le;
  TDKDataSet ds;

  // define other program variables
  unsigned int value;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the “ADDR” label which is found in the dataset
  err = le.attach(ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Retrieve each piece of data and send it to the output
  // window for display
  while (le.next( value ))
  {
    io.printf(“%0X”, value);
  }

}
```

3

Tool Development Kit Interface

The Tool Development Kit tool allows access to the internal data of the logic analyzer. The data can be retrieved and manipulated by means of a programming language.



/hplogic/demo/ToolDevKit/sample1.c (read only)

```
/*  File:  sample1.c
    Purpose:  A simple TDK program to print ADDR values
    to the screen
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  // define LabelEntries and DataSets
  TDKLabelEntry le;
  TDKDataSet ds;

  // define other program variables
  unsigned int value;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the "ADDR" label which is found in the dataset
  err = le.attach(ds, "ADDR" );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
```

Line 1

Col 1

Tool Development Kit Tool Display

The tool consists of four areas: menu bar, tool bar, compile/execute buttons, and a folder with three tabs.

Menu Bar

The menu bar consists of seven options: file, window, edit, view, options, search, and help.

File

Load Configuration. The state of the Tool Development Kit tool (i.e. editor options, file being edited, etc.) can be saved into Tool Development Kit configuration files. These files can be loaded into the tool so that a previous editing session can be restored. When this option is selected, an OpenFile dialog is displayed so that a Tool Development Kit configuration file can be selected and loaded. By default, it has a file extension of ___ (for example, myconfig.___).

Save Configuration. The current state of the Tool Development Kit tool (i.e. editor options, file being edited, etc.) is saved to a configuration file. When this option is selected, a SaveFile dialog is displayed. Enter in the desired name of the Tool Development Kit configuration and select the Save button.

New Source File. This option causes the contents of the editor to be discarded and a new temporary file to be created. If the editor contents have been modified and not saved and this option is selected, a warning dialog will be displayed to confirm the action.

Open Source File. A previously saved source file can be loaded into the editor. The editor contents will be replaced with the new source file. A file selection dialog will appear that allows the desired source to be selected and loaded. This option does not change any editor configuration options--it only loads in a source file.

Insert Source File. Another file can be inserted into the editor buffer at the current cursor location. A file selection dialog will appear that allows the desired file to be selected and inserted. Inserting a file does not change the name of the file being edited.

Reload Source File. The contents of the editor buffer are refreshed by re-reading the disk file.

Save Source File. The contents of the editor buffer are saved to the file name displayed in the editor window. If a temporary source file name is being used, the SaveAs file dialog will appear so that the file can be renamed to something more meaningful.

Save Source File as. The contents of the editor buffer are saved to the named file. The SaveAs file dialog is displayed and allows a file name to be entered. The contents of the buffer will be saved to that name and the editor window will be updated to reflect the new file name.

Create Installable Tool... This option allows a stand-alone tool to be created from the code you have written in the Tool Development Kit. An icon for the tool can be selected and information can be provided in the “Tool Info” tab. It will create a file on the hard disk, or use the floppy disk as the location of the tool. If more than one floppy is required, you will be prompted to enter them in succession. All of the floppy disks will be re-formatted. Once this tool has been created, it may be installed on any Agilent Technologies 16600/700 analyzer using the System Admin Software install menu.

NOTE: Installable tools created on a floppy disk MUST be installed from a floppy disk. Installable tools created on the file system (hard disk or mounted file system) MUST be installed from the file system.

Print options. The Tool Development Kit display can be sent to a printer or to a file. This option does not allow the source file currently being edited to be sent to a printer or to a file. This is only a graphics dump of the window.

Print this window. Using the options set up in the Print Options dialog, the graphic display content of the window is dumped to a file or printer.

Close. Pops down the Tool Development Kit display.

Window

One way to navigate around the workspace is to use the Window menu. Any icon that is on the workspace will have an entry in the Window menu. Selecting an icon name from the menu, displays the icon's window.

Edit

Undo. Reverse the previous editing action.

Redo. Reverse the previous undo action. The contents of the editor buffer will appear as it did before the undo.

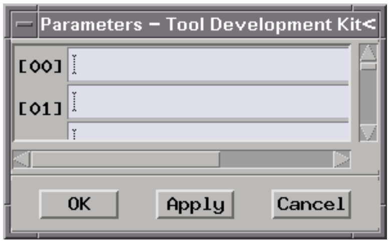
Cut. The selected text is removed and placed on the editor's clipboard.

Copy. The selected text is copied on to the editor's clipboard.

Paste. The contents of the editor's clipboard is copied into the editor's buffer at the current cursor location.

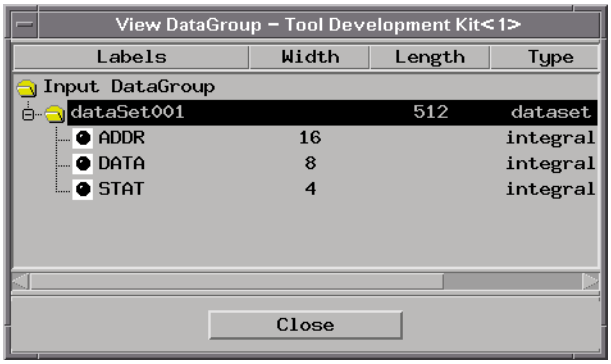
View

View Parameters. The dialog that is displayed allows information to be passed into the tool at runtime. By default there are 50(0-49) lines; however, the actual number parameters can be specified by the program. See "Using Parameters" on page 189. Each line is a free format text area and the information that is entered can be accessed in the tool using the io.getArg() function.



Tool Development Kit Parameters Dialog

View datagroup. The contents of the input datagroup, are displayed in a hierarchical format. This dialog is used for reference information only when developing tools. Below is the view datagroup dialog.



Labels	Width	Length	Type
Input DataGroup			
dataSet001		512	dataset
● ADDR	16		integral
● DATA	8		integral
● STAT	4		integral

Tool Development Kit View DataGroup Hierarchy Dialog

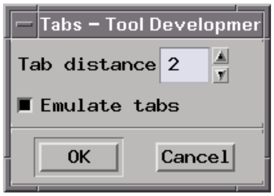
Options

Auto indent [ON/off]. The editor keeps a running indent. When the return key is pressed, spaces and/or tabs are inserted to line up the insert point under the start of the previous line.

Show matching [on/OFF]. Automatic parenthesis matching is activated when you type or move the insertion cursor after a parenthesis, bracket, or brace. It momentarily highlights the matching character if that character is visible in the window.

Tabs. The tab character is inserted into the editor buffer when the tab key is pressed. The tab distance determines how the tab character is to be interpreted when it is displayed. If a space, instead of a tab, is desired, the “Emulate tabs” option (Default: Enabled) is available. The tab key will insert the correct number of spaces to bring the cursor to the next emulated tab stop. Backspacing immediately after entering an emulated tab will delete it as a unit, but as soon as you move the cursor away from the spot, the editor will forget that the collection of spaces represent a tab and it will be treated as separate characters.

To enter a real tab character with “Emulate tabs” on, use CTRL+Tab.



Tab distance 2

■ Emulate tabs

Tool Development Kit Tab Dialog

Overstrike [on/OFF]. Toggle the insertion mode from character insertion to character overwrite.

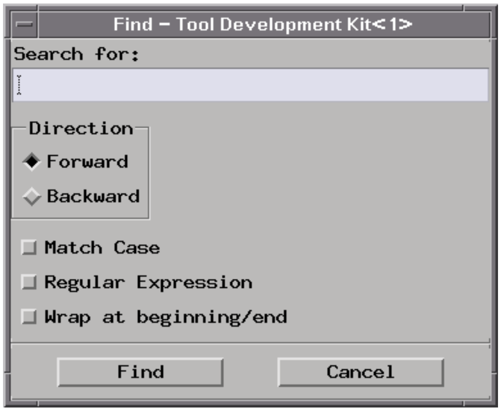
Use external editor [on/OFF]. The file that is currently being edited can automatically be reloaded when the “Run” button is pressed. When the “Run” button is pressed, and this feature is enabled, the time and date of the file being edited are compared to the time and date of the corresponding disk file. If the disk file is newer, it will be loaded into the editor, compiled and executed. This feature allows the disk file to be updated via an NFS mount, because the code is being developed outside of the Tool Development Kit editor, and the “Run” button is the signal that allows the new file to be read in and executed.

This feature can be used in conjunction with the analyzer’s Windows 95/98/NT filesystem connectivity options. In this case, the best approach is to share a drive on the analyzer to the PC. Once the file is visible on the Windows PC, it can be opened and modified with any editor. Remember that the file must be saved on the PC before going back to the analyzer and executing the Tool Development Kit program.

It is inadvisable to map a 95/98/NT network drive on the analyzer for a couple of reasons. One is that the compiler generates a number of intermediate files in the same directory as the source file, and if these are located across a network connection, it can slow compilation time considerably. The other consideration is that DOS files add in a carriage return character that is displayed in the Tool Development Kit editor window. This does not cause any problem in terms of compilation, but is does make for a strange looking display.

Search

Find. A dialog is displayed for entering text for searching. It also allows the choices for search direction, case sensitivity, standard Unix pattern matching characters (regular expressions), and wrapping the search. Searches begin at the current text insertion position.



Find – Tool Development Kit<1>

Search for:

Direction
● Forward
○ Backward

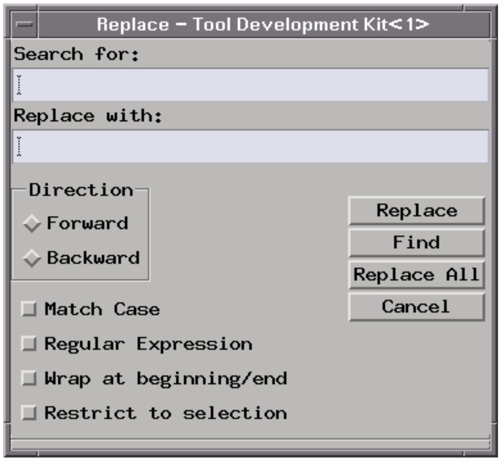
□ Match Case
□ Regular Expression
□ Wrap at beginning/end

Tool Development Kit Search Dialog

Find next. Repeat the last search command without prompting for a search string.

Replace. A dialog is displayed for entering text for searching and replacing. It also allows the choices for search direction, case sensitivity, standard Unix pattern matching characters (regular expressions), wrapping the search/replace, and only replacing in a selected area. Searches begin at the current text insertion position. A combination of “Find” and “Replace” allow selective replacing. “Replace All” will change all occurrences without prompting.

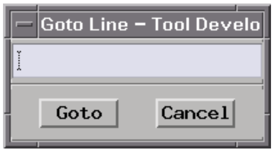
To restrict replacing to a specified area, highlight the area.



Replace Dialog

Replace next. Repeat the last replace command without prompting for a new search/replace string.

Goto line. A dialog is displayed which allows a line number to be entered. The desired line will be moved into the current editor window and highlighted.



Goto Line Dialog

Tool Bar

This area consists of run/stop buttons, window shortcut buttons, and a status line.

Run/Stop Buttons

The Run button causes new data to be acquired and the Tool Development Kit tool to be executed. When a tool is being executed, the Stop button becomes active. Pressing the Stop button aborts the currently executing code if it has been instrumented with calls to io.checkForUserAbort(). All changes that have been made to data are lost.

Window Shortcut Buttons

These buttons let you open the System, Workspace, Intermodule, Run Status, and System Administration windows.

Status Line

System messages are displayed in this area. While a tool is executing, a message will be displayed indicating its progress.

Compile and Execute Buttons

The button bar currently consists of two buttons: compile and execute.

Compile Button

The code that is being edited is compiled. If the code is modified and has not been saved before it is compiled, a dialog will pop up asking whether the source code should be saved. It must be saved before the compile can continue. Any errors that occur during compilation will be displayed in the “Buildtime” window. By selecting the error message, the editor will be positioned to that line.

Execute Button

The code that is being edited is compiled, if necessary, and executed if there are no compilation errors. All incoming data is made available to the tool. The parameters are also made available. The execute button does not cause any new data to be acquired. It only processes the incoming data that is currently available to the tool. If the code is modified and has not been saved prior to executing, it will need to be saved. Any errors that occur during compilation, or execution, will be displayed in the “Buildtime” or “Runtime” windows. By selecting the error message, the editor will be positioned to that line.

Source Code Tab

The Source Code tab is divided into two parts: the title area and the editing area.

Title area

The title area consists of the file name and a status message. When the Tool Development Kit tool is first displayed, a temporary file is created. The tool requires that a valid file be available, hence, a valid temporary file is created. By default, the temporary file is created in /tmp directory. The filename is tmp.c. When a “Save as” is performed, the temporary will be deleted. If the temporary file is empty and an “Open” is performed, the temporary file will be deleted. The status area next to the filename indicates if the file has been modified or is in read-only mode.

Editing area

The editor provides basic mouse and keyboard functions.

Mouse

Mouse Buttons

Click	Left Button	Cursor position and primary selection
Double Click	Left Button	Select a whole word
Triple Click	Left Button	Select a whole line
Shift Click	Left Button	Adjust (extend or shrink) the selection, or if there is no existing selection, begins a new selection between the cursor and the mouse
Ctrl+Shift+Click	Left Button	Adjust (extend or shrink) the selection rectangularly
Drag	Left Button	Select text between where the mouse was pressed and where it was released
Ctrl+Drag	Left Button	Select rectangle between where the mouse was pressed and where it was released
Click	Middle Button	Copy the primary selection to the clicked position
Shift+Click	Middle Button	Move the primary selection to the clicked position, deleting it from its original position
Drag	Middle Button	Outside of the primary selection: Begin a secondary selection. Inside of the primary selection: Moves the selection by dragging
Ctrl+Drag	Middle Button	Outside of the primary selection: Begin a rectangular secondary selection. Inside of the primary selection: Drags the selection in overlay mode.

Moving the Primary Selection by Dragging with the Middle Button

Shift	Leaves a copy of the original selection in place rather than removing it or blanking the area
Ctrl	Changes from insert mode to overlay mode (normally, dragging moves text by removing it from the selected position at the start of the drag, and inserting it at a new position relative to the mouse. When you drag a block of text over existing characters, the existing characters are displaced to the end of the selection. In overlay mode, characters which are occluded by blocks of text being dragged are simply removed. When dragging non-rectangular selections, overlay mode also converts the selection to rectangular form, allowing it to be dragged outside of the bounds of the existing text.).
Escape	Cancels drag in progress

When the mouse button is released after creating a secondary selection

No Modifiers	If there is a primary selection, replaces it with the secondary selection. Otherwise, inserts the secondary selection at the cursor position.
Shift	Move the secondary selection, deleting it from its original position. If there is a primary selection, the move will replace the primary selection with the secondary selection. Otherwise, moves the secondary selection to the cursor position.

Keyboard

Keyboard Actions

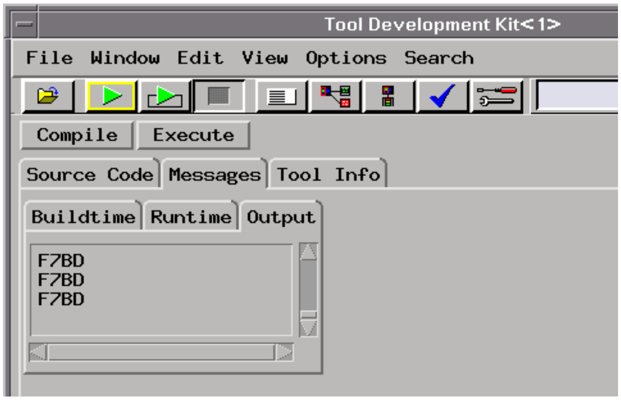
Backspace	Delete the character before the cursor
Left Arrow	Move the cursor to the left one character
Ctrl+Left Arrow	Move the cursor backward one word
Right Arrow	Move the cursor to the right one character
Ctrl+Right Arrow	Move the cursor forward one word
Up Arrow	Move the cursor up one line
Ctrl+Up Arrow	Move the cursor up one paragraph (Paragraphs are delimited by blank lines)
Down Arrow	Move the cursor down one line
Ctrl+Down Arrow	Move the cursor down one paragraph
Ctrl+/	Select everything
Ctrl+\	Unselect everything
Delete	Delete the character before the cursor
Ctrl+Delete	Delete to end of line
Shift+Delete	Cut, remove the currently selected text and place it in the clipboard
Home	Move the cursor to the beginning of the line
Ctrl+Home	Move the cursor to the beginning of the file.
End	Move the cursor to the end of the line.
Ctrl+End	Move the cursor to the end of the file
PageUp	Scroll and move the cursor up by one page
PageDown	Scroll and move the cursor down by one page.

Status

At the bottom of the Source Code tab there is a small status line. The current cursor position, and the line and column are displayed. There is also a small message area where the editor puts informative messages.

Messages Tab

There are three windows: Buildtime, Runtime, and Output.



Tool Development Kit<1>
File Window Edit View Options Search
Compile Execute
Source Code Messages Tool Info
Buildtime Runtime Output

F7BD
F7BD
F7BD

Buildtime

Compilation messages are sent to this window. When the compile or execute button is pressed, the contents of this window are deleted. When an error message is displayed, the line in the code causing the error can quickly be brought into the edit window by selecting the error message.

Runtime

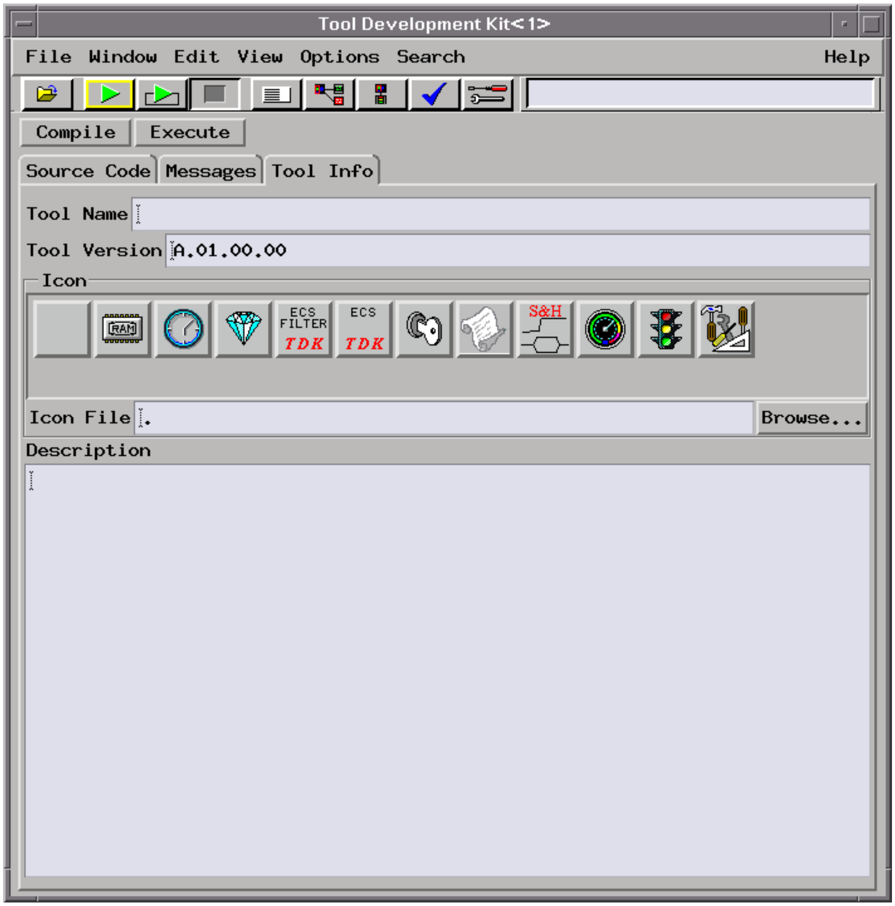
All runtime errors, such as failures to attach to label entries and data sets, will be displayed in this window. When the compile or execute button is pressed, the contents of this window are deleted.

Output

The Tool Development Kit tool has built-in output capabilities (print). The output generated by the tool will be placed in this window. When the compile or execute button is pressed, the contents of this window are deleted. Please note excessive writing to this window can cause the Tool Development Kit tool to perform poorly. The data from this area can be cut and pasted into other windows. Triple clicking in this window selects all of the data, not just the displayed, which can then be pasted into another window.

Tool Info Tab

The tool info tab allows a tool developer to give a name and icon to the tool.



Tool Name

Tool Version A.01.00.00

Icon File .

Tool Development Kit Tool Info window

This window should be filled out before creating an installable tool. The icon browse area allows an icon to be associated with the tool. This will appear on the workspace once the tool is installed on the analyzer. Tool Name is what the tool will be called. Additional information can be general information about the tool or the developer of the tool. It will be visible to the end user while the tool is being installed.

4

Tool Development Kit Concepts

This chapter covers concepts fundamental to using the Tool Development Kit tool to write programs that manipulate data of interest.

To successfully create programs using the Tool Development Kit an understanding of the different types of data and how they are accessed is first required.

Examples will be presented to help show how to access data in the sections Working with TDKDataGroups, Working with TDKDataSets, Working with TDKLabelEntries and Working with TDKCorrelators.

Types of Data

There are three types of data supported by the Agilent Technologies 16600 and Agilent Technologies 16700: Integral, Analog, and Text data. It is important to know the type of data that has been captured by the logic analyzer and passed into the Tool Development Kit tool because it determines which specific Tool Development Kit library function should be used to access the data. For example, if Analog data has been captured, then the Analog version of library functions should be used to access the Analog data. Using an Integral or Text version of the library functions to access Analog data will not result in correct retrieval of the data.

Integral Data

Integral data is numeric data. It is an unsigned quantity ranging from 1 to 32 bits in width. When data is acquired with one of the acquisition modules (Agilent Technologies 16555/16550, etc.), the data is transformed into Integral data. One exception to Integral data occurs with accessing a time value or state value as opposed to the actual data value captured for a state or time period. In the time or state case, the Integral data is a signed 64-bit quantity. Integral data is the most common type of data captured. Unless you are using an oscilloscope module, you can assume the data passed into the Tool Development Kit tool is Integral data. In general, Tool Development Kit library functions used to access data that contain a parameter of type "unsigned int" are to be used to access Integral data.

The following table shows an example of a listing where Integral data has been captured. State information and time information are stored in a signed 64-bit quantity. Actual data values that are found under the ADDR label are stored in an unsigned quantity of 16 bits for this example. In general, the size for any given label entry, such as ADDR, can be retrieved by referencing the Machine Format dialog and looking up the label name.

Sample Trace Listing

State Information (signed 64 bit value)	Timing Information (signed 64 bit value)	ADDR Label (unsigned 16 bit value)
0	1.0 ns	0x1234
1	3.0 ns	0x5678
2	5.0 ns	0x1212
3	7.0ns	0x3434

Analog Data

Analog data is stored as a 15-bit integer, but is accessed like a double. It is meant to represent data coming from an oscilloscope, so to represent Analog data, an offset and full scale volts per screen must be given. In general, Tool Development Kit library functions used to access data that contain a parameter of type "double" are to be used only for oscilloscope captured data.

Text Data

Text data is ascii data. Text data is useful when you want to display information that more fully explains a condition. For example, assume that setup and hold time measurements are being made and that a particular piece of data violates the hold time. A piece of text data can be created which says, "Hold time violation". This text data will be displayed in the Listing window making it easy to locate the violations.

In general, text data will not be passed in to the Tool Development Kit tool. Rather a Tool Development Kit program can be used to create text data that becomes an output of the Tool Development Kit. This output usually serves as input to the lister tool or some other tool.

The exception to this is if a Tool Development Kit program creates Text data and passes that data into some other Tool Development Kit tool or program. Tool Development Kit library functions used to access data containing a parameter of type "String" are used for Text data.

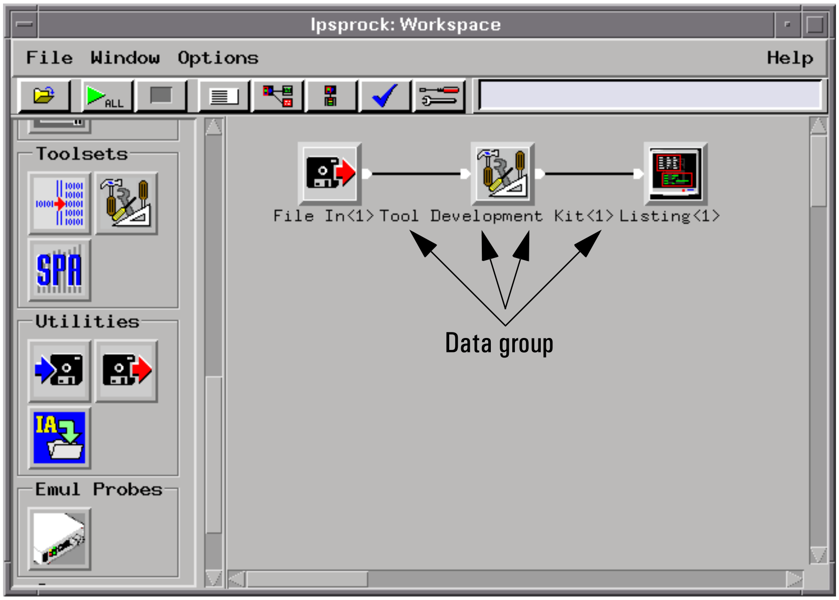
For information on accessing data using Tool Development Kit library functions, see the section "Using Iterators to Access Data" on page 59. The table below shows a summary of the different data types together with the modules used to capture the data. Note, the Tool Development Kit, while not a measurement module, can be used to output new data of any given data type (Integral, Analog, or Text). The measurement modules in the table below can only output Integral or Analog data. See the section on creating a new TDKLabelEntry on page 101 for more information as to how this is done.

Data Types Output by Various Measurement Modules

Measurement Module Category	Integral Data	Analog Data
State and Timing Acquisition Cards (for example 16550A, 16557A/D, 16710A, 16715A, etc.)	X	
Oscilloscopes (16533A/16534A or later)		X
High-Speed Timing (16517A/16518A or later)	X	
Pattern Generator (16522A) *	X	

* The pattern generator generates Integral data that does not contain any timing information. It contains state information only.

Data Organization



Simple Workspace Layout with Data Group Points Highlighted

Data groups

When an instrument acquires data, it formats the acquired data into a data group. This data group is passed to the input of the Tool Development Kit program. The program retrieves the necessary information from the data group in order to process it. Once it has processed the designated data, it passes the data group to the input of the listing tool. The listing tool retrieves the information that it needs from the data group in order to update the display.

The data group is similar to a database-- it is the general container where all data resides. At every input/output point, there is exactly one data group that is passed into the tools. If there are multiple "inputs" attached to a single tool (fan-in), all of the data groups are consolidated into a single entity so that the tool receives one, and only one data group. The data group represents the global entity that contains all acquired data. It is further broken down into data sets.

Data sets

A data set contains a group of labels that were defined in the instrument's Format dialog. If you have multiple instruments (measurement modules) fanned-in to a single tool, there will be a data set in the data group for each instrument. The data set represents how the instrument acquired the data (number of samples, acquisition time, acquisition rate) and it also holds the name of the acquisition instrument. All labels defined within a data set have the same sampling information (i.e. number of samples acquired, sample rate, acquisition time).

Label entries

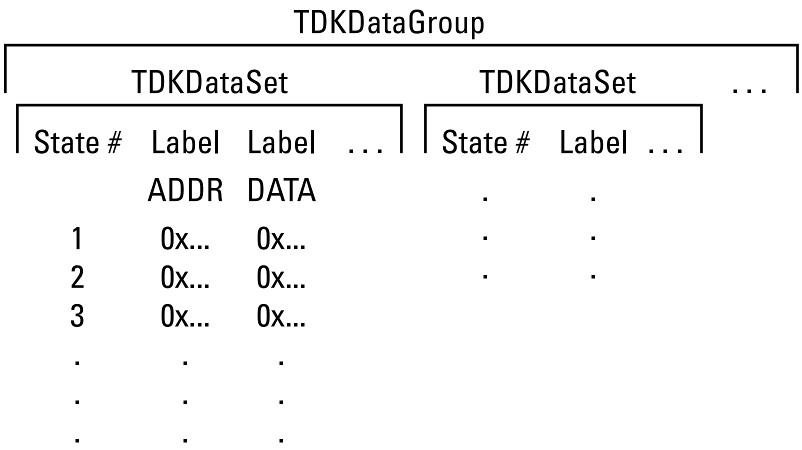
Each label within a data set corresponds to the label that was designed in the instrument's Format dialog. A label, within a data set, consists of its name, width (number of bits assigned to the label), and its type (whether it is integral, analog, or text). A label also holds all of the data that was acquired. Therefore, in order to access acquired data, it is important to know the name of the label, the name of the instrument where that label is defined, and the type of data captured (integral, analog, or text). These pieces of information are critical for data retrieval.

Tool Development Kit library functions refer to each of these three data structures by the following data type names:

- data group == TDKDataGroup
- data set == TDKDataSet
- label or label entry == TDKLabelEntry

These names will be used throughout this manual.

Accessing the Data

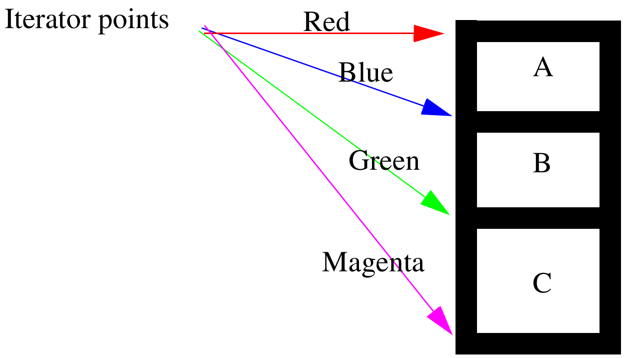


Data Hierarchy in the Tool Development Kit

The main focus of the Tool Development Kit is to allow users to freely manipulate their data. This includes basic iterators for stepping through the data. It also includes higher-level correlated iterators that handle the most common types of state and timing correlation. These basic means of support provide Tool Development Kit programmers with tremendous flexibility.

Using Iterators to Access the Data

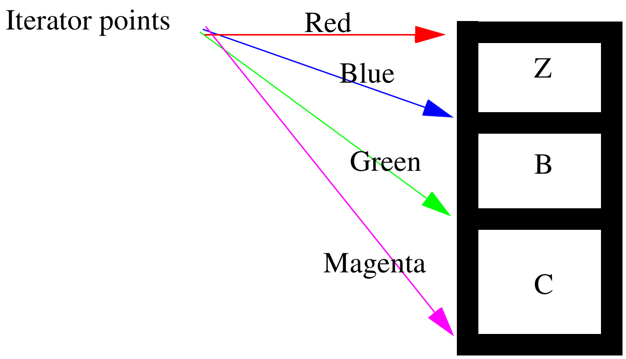
Accessing of the underlying data is handled with iterators. They create a level of abstraction that shields the developer from the data storage mechanism. They also provide a consistent method that is applied to retrieving and storing of data. The general concept is that iterators don’t point at data, they point in between data.



Iterator Diagram Showing “Next” and “Prev” Functionality

Data is retrieved by using "next" or "prev". As the data is stepped over, it is retrieved and returned. As an example, assume that the red line is the iterator of interest. If "next" is used, the cell containing the value "A" is stepped over. The process of stepping over the cell causes the contents of the cell, in this case the value "A", to be retrieved and returned. The iterator is now placed before the next cell--the blue line. If "next" is used again, the value "B" is returned and the iterator is now the green line. If "next" is used again, the value "C" is returned and the iterator is now the magenta line. If "next" is used again, there is not a value returned and the iterator indicates that it is at the end of data. The same analogy is true with "prev." If the starting position is the magenta line and "prev" is used, the value "C" is returned and the iterator is now the green line. If "prev" is used, the value "B" is returned and the iterator is now the blue line. If "prev" is used, the value "A" is returned and the iterator is now the red line. If "prev" is used again, there is not a value return and the iterator indicates that it is at the beginning of data. An iterator can be set to begin iterating at any position within the data.

Data is stored by using the "replaceNext" or "replacePrev" functions. Instead of retrieving the data as it is being stepped over, the data is replaced with the new data as it is being stepped over. As an example, assume that the red line is the iterator of interest. If "replaceNext" is used, and the value "Z" is the new data, the cell containing the value "A" is stepped over. The process of stepping over the cell causes the contents of the cell to be replaced with the new data and the iterator is placed at the beginning of the next cell--the blue line.



Iterator Diagram Showing “A” Being Replaced with a “Z”

This process is repeated to replace all of the data. The function "replacePrev" operates the same as "prev" except that it replaces the data as it is stepped over. These four functions constitute the interface into the underlying data. Once they are understood, all data can be accessed, manipulated and stored.

Understanding Correlation

Correlation refers to how data acquired from multiple probe sources (measurement modules) should match up when iterating through the data. Possibilities include State, Timing, both State and Timing, or no correlation. Correlation is guaranteed if the incoming data group contains only one data set.

Time Correlation is required when there are multiple analyzers acquiring data on independent time-bases. In this case the samples from one analyzer will need to be "correlated" to those of another to ensure that data from the same instant in time is being compared. This may be achieved by writing the code to manage the timing information, or the built-in Tool Development Kit Correlators can be used. It depends upon the particular application.

An upstream tool called a Pattern Filter may have filtered the data that comes into the Tool Development Kit tool. Data filtration occurs at the data set level. That is, if a particular sample is filtered out because it matches the criteria specified in the pattern filter, it is eliminated from every label entry within that particular data set. This detail is only important when the tool is trying to analyze more than one data set. If there is more than one data set in the Tool Development Kit, frequently it is desired to have the samples of one data set "match up" with the samples of another. This situation is called *correlated iteration*.

The key to making this match-up work for any particular time instant is to retrieve data only from those labels that were not filtered at that time. To find out whether a particular sample exists and was not filtered, functions of the data set (see the section Working with TDKDataSets on page 65) must be called that return the exact position of the next (unfiltered) sample. This position can then be used by each label entry to set its position. For each one that does exist, the data within the label entry at that sample should be retrieved.

Working with TDKDataGroups

The collection of all data that is passed into any tool, including the Tool Development Kit, is called a data group. A TDKDataGroup is passed into the "execute" routine by reference. This variable, *dg*, should be used with all of the TDKDataGroup functions. Tool Development Kit programs will never have a need to declare variables of type TDKDataGroup.

A Tool Development Kit data group is a collection of all the data sets present in the run, along with correlation information about those data sets. A data group will always contain at least one data set. The correlation information tells how the data sets in a data group are related. Recall that data sets can be correlated in State, Time, both State and Time, or no correlation. A data group with a single data set that has State and Time information will be State and Time correlated.

Data group information can be obtained non-programmatically by viewing the Tool Development Kit menu option View datagroup.... This dialog shows information pertaining to the incoming data group *dg* that is passed into the Tool Development Kit tool "execute" function. During development of a Tool Development Kit program, this information can be most helpful as it provides a list of all the data sets contained within the captured data group. For each data set listed, the name of the data set is shown together with all the label entries contained in each data set. For each label entry listed, the name of the label entry, size of the label entry and type of data is shown. While all of this information can be obtained programmatically through Tool Development Kit library functions, this dialog can be useful to show how the captured data is organized.

TDKDataGroup Functions

dg.getNumberOfDataSets

```
int dg.getNumberOfDataSets()
```

This function returns the number of data sets present in the data group.

dg.getDataSetNames

```
int dg.getDataSetNames( StringList& names)
```

This function returns the number of data sets present in the data group and also puts the names into the StringList names that is passed as a parameter. names is reSize()'d to the number of data sets.

```
int i;
StringList names;
dg.getDataSetNames(names);
for(i = 0; i < dg.getNumberOfDataSets(); i++)
{
   io.print( names[i] );
}
```

dg.isTimeCorrelatable

```
int dg.isTimeCorrelatable()
```

This function returns true if the data group is time correlatable.

dg.isStateCorrelatable

```
int dg.isStateCorrelatable()
```

This function returns true if the data group is state correlatable.

dg.setTimeCrossCorrelation

```
int dg.setTimeCrossCorrelation()
```

This function should be called in a multiple data set situation to tell the system that they should be correlated by time. Returns an error code.

dg.setStateCrossCorrelation

```
int dg.setStateCrossCorrelation()
```

This function should be called in a multiple data set situation to tell the system that they should be correlated by state. Returns an error code.

NOTE: It is important that either dg.setTimeCrossCorrelation or dg.setStateCrossCorrelation be called any time new data sets have been created through the Tool Development Kit tool environment. This informs downstream tools that either time or state correlated data is available.

dg.removeDataSet

```
int dg.removeDataSet(TDKDataSet ds)
```

This function removes the data set that *ds* is attached to. Returns 1 for success, 0 for failure. This function is useful in situations where a new data set is to be displayed without showing the original data set. The original data set can be removed by using this function and passing in the data set reference *ds*.

Working with TDKDataSets

It is possible to create a new data set. This is often useful if the information that is to be created will not have the same number of samples as the input. For example, if you are creating a tool that takes in serial words and outputs parallel words, then the output data set will have only 1/8th the number of samples as the input data set, assuming that the bits are being assembled into bytes.

On the other hand, a tool that is designed to demux an incoming data set, will need to create a data set that has more samples than the input based on the mux factor. If the mux factor is 4, then the output data set will need 4x number of samples.

If it is not easy to determine the number of samples that the output data set will need, (i.e. there is no easy calculation which can give this result) then you will need to create one that is guaranteed to be big enough, even if it is too big. This is because you cannot change the size of the data set once you have created it. At the end, it is easy to filter any extra states with the ds.filter( ) function to hide them from the user.

Once you have decided how big to make the output data set, then you must create labels within that data set. See the section on Working with TDKLabelEntries on page 101 for details. Also, you will probably want the input samples to "correlate" to the output samples. This can be done by modifying the time stamps on the output data set. Using the ds.replaceNext() function, you can read time stamps from the input and write them to the output. When the Listing tool displays the information, it aligns all the samples according to their time stamps.

NOTE: It is not possible to change the number of samples that a data set holds once created; this is fixed.

Data sets are declared simply as:

TDKDataSet ds;

Where *ds* is the data set variable being declared. This does not initialize the variable. Exactly one of the attach() or create() functions must be called to *instantiate* the variable before it can be used in any other operations.

TDKDataSet Creation Functions

When creating new TDKDataSets, you must be aware of the kind of correlation that is available, if you need them to be correlated. Recall that correlation can be State, Timing, both State and Timing, or no correlation. Additionally, you may be forced to remove() one of the other data sets to achieve the correct correlation.

NOTE: Use the data group functions “isStateCorrelatable” and “isTimeCorrelatable” to determine if the input data set(s) can be correlated to newly created data sets in time, state, both time and state, or no correlation. Agilent Technologies 1660X analyzers only produce time correlatable data sets. It does not make sense to attempt to correlate data sets based on state information or use the createState function for creating new data sets on those analyzers.

Most of the information needed to create a new data set can be retrieved from one of the original data sets contained in the data group passed into the execute function. Depending upon your application, you may want this information to be the same or not. The original data group *dg* passed into the execute routine is passed into all data set creation functions in order to know the data group this data set is being created for.

ds.createTimeTags

There are times when it would be convenient to add a new data set, and new label entries within that data set. The new data set might have a different number of samples and sample frequency than the original data set and it is desired that the new data set allow the time of each sample to be modified. By modifying the time stamps of the newly created data set, user-specified correlation with the incoming data is possible. The function createTimeTags should be used to create a data set with these desired properties. The majority of applications that create a new data set will use this function. If you are not sure which data set creation function to use, use this one.

```
int ds.createTimeTags(TDKDataGroup& dg,
String name,
unsigned int len,
unsigned int triggerRow,
long long correlationTimeOffset,
long long samplePeriod)
```

This function creates a new data set *ds* called *name* with time tags and *len* number of samples. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationTimeOffset* tells how the data set trigger position matches up in time in case there is more than one data set. The *samplePeriod* tells the time instants of each of the samples. Returns an error code.

```
TDKDataSet originalDS;
TDKDataSet newDS;
originalDS.attach(dg);
newDS.createTimeTags(dg, "newDataSet", 100, 50,
originalDS.getCorrelationTime(), nanoSec(4.0));
```

NOTE:

When creating a new data set(s) with time tag information, pass in the correlation time offset retrieved from one of the data sets inputted to the Tool Development Kit tool. In general, all time correlated data sets output from the Tool Development Kit tool need to have the same correlation time offset value so that they are properly aligned in downstream tools such as the Listing tool.

Also, note that anytime you create new data sets that are correlated in time, you must call “dg.setTimeCrossCorrelation” before you exit the “execute” function. See the tutorial “Creating a new data set with modifiable time stamps” on page 94 for an example.

In addition, the following rule must be followed when creating a data set for which time stamps will be modified.

RULE:

Each subsequent sample in a data set must contain a monotonically increasing time stamp.

For example, it is incorrect to create a data set with modifiable time stamps (using the createTimeTags function) and give sample 10 a time stamp of 400 nanoseconds while sample 11 has a time stamp of 100 nanoseconds.

A common programming error is to create a new data set using the createTimeTags function and not adhere to this rule. This can easily happen in the following scenario.

Suppose an incoming data set contains 100 samples. Your program also creates a new data set with 100 samples. Next your program iterates through the incoming 100 samples and finds the starting states of some transaction of interest. Whenever a new transaction state is found, the time stamp from the state at which it is found in the incoming data set is used to replace the time stamp for a sample in the new data set, using the data set "replaceNext" function call. Your algorithm determines that there are 10 such starting transaction states in the incoming data set. Thus, your algorithm only replaces the time stamps on the first ten samples in the new data set while leaving the remaining 90 time stamps unmodified. This has the potential to create undefinable behavior in the correlation of the data sets in the Listing tool. Here's why:

One side effect of using the createTimeTags function is to assign an initial sample period to each sample in the new data set. These initial sample time stamps will always adhere to the rule of data sets containing monotonically increasing time stamps.

For example, if you used the function

```
newDS.createTimeTags(dg, "newDataSet", 100, 0,
originalDS.getCorrelationTime(), nanosec(4.0) );
```

the first sample in the data set will have a time stamp of 0.0 nanoseconds, the second will have a time stamp of 4.0 nanoseconds, the third will have a time stamp of 8.0, and so forth up to the last sample having a time stamp of 400 nanoseconds.

You then replace the first ten time stamps in your new data set with increasing values (i.e. 0.0, 1.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0 nanoseconds). You don't replace the time stamps on samples 11 through 100, so they are still set to the value assigned to them during the creation of the data set (namely 44.0, 48.0, 52,0, and so on up to 400.0 nanoseconds). The rule of having monotonically increasing time stamps in any given data set has just been violated because sample 10 has a time stamp of 80.0 nanoseconds while sample 11 has a time stamp of 44.0 nanoseconds. The analyzer cannot correlate or display data accurately in this manner.

There are two ways to achieve the desired results in this scenario. One way is to keep track that you have only modified 10 of the new data sets time stamps and then fill the remaining 90 samples with dummy values (that ensures monotonically increasing time stamps). Then you can filter out these 90 states using the data set filter function described in the section "Filtering data within the Tool Development Kit". Another way is to iterate through the incoming data set first and determine that you only need to create a data set with ten samples. Then create a data set with exactly the number of samples that you will need to modify. This will negate having to create dummy values and then filtering those dummy samples.

By adhering to these practices, you will save yourself a lot of time trying to figure out why your program is not working as intended.

There are times when it may be useful to create a copy of the same data set. This is achieved by using the overloaded createTimeTags function passing in the data set to be copied as the third parameter.

```
int ds.createTimeTags(TDKDataGroup& dg,
String name,
TDKDataSet origDS,
unsigned int triggerRow,
long long correlationTimeOffset,
long long samplePeriod)
```

This function creates a new data set *ds* called *name* which is a copy of *origDS* with timeTags added. *ds* will contain all the label entries of *origDS*. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationTimeOffset* tells how the data set trigger position matches up in time in case there is more than one data set. The *samplePeriod* tells the time instants of each of the samples. Returns an error code.

```
TDKDataSet ds;
TDKDataSet orig;

orig.attach( dg );

ds.createTimeTags( dg, "newDataSet", orig,
orig.getTriggerRow(), orig.getCorrelationTime(),
nanoSec(4.0) );
```

ds.createState

Sometimes it may be useful to create a new data set that contains only state information and no timing information. Most likely this will useful if doing a one to one correlation with another data set. This can be achieved by using the *createState* function.

```
int ds.createState( TDKDataGroup& dg, String name,
unsigned int len,
unsigned int triggerRow,
long long correlationStateOffset)
```

This function creates a new data set *ds* with state information called *name* and with *len* number of samples. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationStateOffset* tells how the data sets trigger position matches up in time in case there is more than one data set. Returns an error code.

```
TDKDataSet originalDS;
TDKDataSet newDS;

original.attach(dg);
newDS.createState(dg, “neDataSet”, 100, 50,
originalDS.getCorrelationState());
```

NOTE: When creating a new data set(s) with state only information, pass in the correlation state offset retrieved from one of the data sets input to the Tool Development Kit tool. In general, all state correlated data sets output from the Tool Development Kit tool need to have the same correlation state offset value so that they are properly aligned in downstream tools such as the Listing tool.

Also, note that anytime you create new data sets that are correlated in state, you must call “dg.setStateCrossCorrelation” before you exit the “execute” function.

ds.createTimePeriodic

There are times when it would be convenient to add a new data set, and new label entries within that data set where the new data set can have a different number of samples and a different constant sampling period than the incoming data set. This is achieved by using the createTimePeriodic function.

```
int ds.createTimePeriodic( TDKDataGroup& dg,
String name,
unsigned int len, unsigned int triggerRow,
long long correlationTimeOffset,
long long samplePeriod)
```

This function creates a new data set called *name* with time period information and *len* number of samples. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationTimeOffset* tells how the data set trigger position matches up in time in case there is more than one data set. The *samplePeriod* tells the time instants of each of the samples. Returns an error code.

```
TDKDataSet originalDS;
TDKDataSet newDS;

originalDS.attach(dg);
newDS.createTimePeriodic(dg, “newDataSet”, 100, 50,
originalDS.getCorrelationTime(), nanoSec(4.0))
```

NOTE: When creating a new data set(s) with time tag information, pass in the correlation time offset retrieved from one of the data sets inputted to the Tool Development Kit tool. In general, all time correlated data sets output from the Tool Development Kit tool need to have the same correlation time offset value so that they are properly aligned in downstream tools such as the Listing tool.

Also, note that anytime you create new data sets that are correlated in time, you must call “dg.setTimeCrossCorrelation” before you exit the “execute” function. See the tutorial “Creating a new data set with modifiable time stamps” on page 94 for an example.

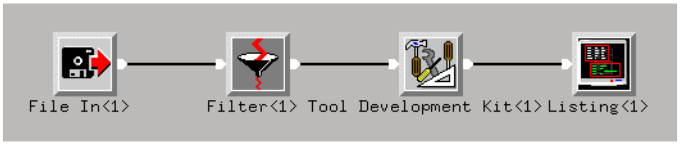
NOTE: It is important that either dg.setTimeCrossCorrelation or dg.setStateCrossCorrelation be called any time new data sets have been created through the Tool Development Kit tool environment. This informs downstream tools that either time or state correlated data is available.

The attach functions shown next are used when it is desired to just associate a variable name with an existing data set. Data sets contained in the original data group passed into the execute function are read-only by default. You cannot change the value of any of the data contained in the label entries associated with the data set. You can however, change the color or highlight the data contained in label entries of read-only data sets.

ds.attach

```
int ds.attach( TDKDataGroup& dg, String name )
int ds.attach( TDKDataGroup& dg )
```

This function associates the variable *ds* to an existing data set called *name* that is present in the current run. It is permissible to re-attach() the same data set variable to another in-coming data set. The effect of this is as if it had never been attach()ed in the first place. Returns an error code.



An example 16700 Workspace

Data set names are constructed by concatenating the names of all instruments and tools through which the data is passed into a string separated by colons. The entire data set name is referred to as the “origin name” while the right-most separated colon is referred to as the “base name”. Recall from page 63 that the data group function “dg.getDataSetNames” returns a string list of the data set names in the data group. This list of names contains the entire name or origin name for each data set. The attach function that takes a String parameter, can be passed either the origin name or the base name of the data set. It will attach to the first data set instance it finds that exactly matches the origin or base name.

The second version of attach(dg) will attach to the very first input data set found in the data group. This allows the programmer to not worry about what the name of the data set is if it does not matter. The “View->datagroup...” dialog will show the order of the incoming data sets along with a substring of the data set name.

For example, suppose we have the following data sets in the incoming data group. The data set names were retrieved programmatically by using the dg.setDataSetNames function.

Data Set #1 => Frame 10: Slot B: Analyzer<B>
Data Set #2 => Frame 10: SlotB: Analyzer <B>_TZ
Data Set #3 => File In<1>: Data Generator<1>: dataSet001
Data Set #4 => File In<2>: Data Generator<2>: dataSet001

To attach to the first data set:

1. ds.attach(dg); or
2. ds.attach(dg, “Analyzer<B>”); or
3. ds.attach(dg, “Frame 10: Slot B: Analyzer<B>”);

To attach the second data set:

1. ds.attach(dg, “Analyzer<B>_TZ”); or
2. ds.attach(dg, “Frame 10: Slot B: Analyzer<B>_TZ”);

To attach the third data set:

1. ds.attach(dg, “File In<1>:Data Generator<1>: dataSet001”);

To attach the fourth data set:

1. ds.attach(dg, “File In<2>: Data Generator<2>: dataSet001”);

NOTE: It is possible to pass in just the base name “dataSet001” to attach to the third data set because the algorithm will attach to the first data set it finds with the passed in origin or base name. However, it will not be possible to attach to the fourth data set if you just pass in the base name “dataSet001”. In this case, the safest bet is to always pass in the entire name of the desired data set.

Two different data set variables can be attached to the same incoming data set. Using this feature, one variable can be used for State based iteration (i.e. iterating through the data set using State information), while the other can be used for Timing-based iteration.

NOTE: There may be more than one data set in the incoming data group. Use the data group function getNumberOfDataSets() to find out how many there are if you are not sure. If this returns more than one, then use the function getDataSetNames to retrieve a list of names of available data sets. Then use the overloaded data set attach function requiring a data set origin name or base name to ensure you attach the local data set to the incoming data set of interest. See the example below for how to do this.

The following example shows how to retrieve the number of data sets in a data group and attach a data set variable to each. For this example, data sets are stored using an array. It is assumed there is no more than 5 data sets in this example.

```
int k;
int err;
int numDataSets;
StringList names;

dg.getDataSetNames( names );
numDataSets = dg.getNumberOfDataSets();
TDKDataSet ds[5];

for (k = 0; k < numDataSets; k++)
{
    io.print( names[k]);
    if ( k <= 4 ) // assume no more than 5 data sets
    {
        err = ds[k].attach(dg, names[k]);
        if (err)
        {
            io.printError( err );
            return;
        }
    }
}
```

ds.isAttached

```
int ds.isAttached()
```

This function returns true if the data set has been attached or created successfully on this run.

TDKDataSet Utility Functions

ds.removeLabelEntry

```
int ds.removeLabelEntry(TDKLabelEntry le)
```

The label entry variable *le* passed in is removed from the data set. Returns the number of label entries found in *ds* with the same name as *le*. This value should be 1, unless for some reason there is more than one label entry with the same name in the data set.

ds.getNumberOfLabelEntries

```
int ds.getNumberOfLabelEntries()
```

This function returns the number of label entries contained in the data set.

ds.getName

```
String ds.getName()
```

This function returns the string name of the data set. This name is a colon-separated list of all the tools that feed into the Tool Development Kit Tool.

ds.getTriggerRow

```
unsigned int ds.getTriggerRow()
```

This function returns the relative sample number of the trigger row. Note this is not given in terms of a state number, as found under the State label in listing tool. Rather this number is in terms of the number of samples in the trace starting with the first sample (which is zeroth based) counting down to the trigger row sample. This is equal to the distance in samples from the first sample of the trace to the sample in the trace containing the trigger row.

For example, if a trace listing contains 1000 samples with the first sample in the trace listing indexed as state -500 and the trigger row is centered in the trace listing indexed at state 0 with 499 samples following the trigger row, then this function will return 500. Taking into account that the samples are zeroth based, the sample in the listing with a state index of -500 is really sample 0, state index -499 is sample 1 and so on. State 499 is sample 1000.

This function is useful when a new data set is created or copied from an existing data set and it is desired for the new data set to have the same trigger row as some original data set.

ds.getBeginTime

```
int ds.getBeginTime(int A[ ])
```

This function changes the given array *A* of integers to a time representing the approximate start time of the run. Returns an error code. The meanings are as follows:

```
A[0] /* years since 1900 */
A[1] /* month of year - [0,11] */
A[2] /* day of month - [1,31] */
A[3] /* hours - [0,23] */
A[4] /* minutes after the hour - [0,59] */
A[5] /* seconds after the minute - [0,59] */
```

ds.getEndTime

```
int ds.getEndTime(int A[])
```

This function changes the given array *A* of integers to a time representing the approximate end time of the run. Returns an error code. The meanings are as follows:

```
A[0] /* years since 1900 */
A[1] /* month of year - [0,11] */
A[2] /* day of month - [1,31] */
A[3] /* hours - [0,23] */
A[4] /* minutes after the hour - [0,59] */
A[5] /* seconds after the minute - [0,59] */
```

ds.getLabelEntryNames

```
int ds.getLabelEntryNames( StringList names )
```

This function returns the number of label entry names present in the data set and fills the given string array with their names.

```
int i;
StringList names;
```

```
ds.getLabelEntryNames(names);
for(i = 0; i < ds.getNumberOfLabelEntries(); i++)
{
    io.print( names[i] );
}
```

ds.getNumberOfSamples

```
int ds.getNumberOfSamples()
```

Returns the number of samples present in the data set. All label entries contained in this data set have this number of samples as well, by definition.

TDKDataSet Filtering Functions

The filtering functions below all work with state, rather than timing information, regardless of the current bias setting.

ds.displayStateNumberLabel

```
ds.displayStateNumberLabel( bool )
```

Disables (or enables) the display of the state number labels for any dataset. If the parameter is set to "false", the state number label for the associated dataset will not appear and will not be available in the listing display. The default value for a dataset is "true" meaning the state number label will appear in the listing display, even though it is not a label that is explicitly created in the TDK code.

ds.filterAllStates

```
int ds.filterAllStates()
```

This function will remove all states as if the Pattern Filter had removed them. Returns an error code.

ds.filter

```
int ds.filter(long long s)
```

This function will remove the given state *s* as if it had been removed by the Pattern Filter. Returns an error code.

```
ds.filter( 56 );
```

This call will filter the state 56 from the data set *ds*. It will not be available in any of the label entries contained in *ds*.

ds.unfilter

```
int ds.unfilter(long long s)
```

If a state has been filtered using the int ds.filter() function, a call to int ds.unfilter(long long *s*) makes the state *s* visible again. This function is not valid for states that have been filtered by the Pattern Filter Tool itself, as the Tool Development Kit tool does not access these states. Returns an error code.

```
ds.unfilter( 56 );
```

This call will make the state 56 available again assuming it was previously filtered in the Tool Development Kit tool.

TDKDataSet Time and State Functions

The data set contains information about time and state. In the listing tool this information shows up in the columns labeled "Time" and "State Number". *Even though these columns appear to be label entries, they are not*. They represent information that is included in the data set.

To access time stamps and state numbers, you must use the iterators (next( ), prev( ), replaceNext( ), etc.) on the data set or label entry.

NOTE:

You can only modify time stamps for data sets that were created using the ds.createTimeTags function. Using the ds.createTimePeriodic function for creating a new data set creates a data set with a constant sampling period and thus the time stamps for such a data set are not modifiable.

Bias Setting

The following TDKDataSet functions are sensitive to the bias setting for a given data set. Recall that the default bias setting is State. Bias settings for a data set can be changed to either State or Time by using either ds.setStateBias or ds.setTimeBias.

- ds.getPosition
- ds.setPosition
- ds.firstPosition
- ds.lastPosition
- ds.next
- ds.prev
- ds.peekNext
- ds.peekPrev
- ds.replaceNext
- ds.replacePrev

For example, if the bias setting for a given data set is time (ds.setTimeBias explicitly called), then the function ds.getPosition will return the value found under the Time column in the listing window. If the bias setting is State (default), then ds.getPosition will return the value found under the State column in the listing window.

The following TDKDataSet functions are used to get various Timing and State information. These functions all return State or Timing information not the actual sample data values.

Each data set variable maintains a pointer to the current sample within the data set. The following functions refer to, or change this pointer.

ds.setTimeBias

```
int ds.setTimeBias()
```

This function sets Timing bias for the data set. Bias (state or timing) indicates the type of information many of the data set functions will operate on. The default bias is State, since timing information may not exist. Returns an error code.

ds.setStateBias

```
int ds.setStateBias()
```

This function sets State bias for the data set. Bias (state or timing) indicates the type of information many of the data set functions will operate on. The default bias is State, since timing information may not exist. Returns an error code.

Note that it is permissible to set the bias to be State and then later change it to Time (and vice versa) for data sets. If preferred, two variables can be attached to the same data set. The bias for one of the variables can be set to State while the bias for the other can be set to Time to achieve the same results.

ds.reset

```
void ds.reset()
```

This function resets the pointer before the first item in the data set. The *ds* pointer will by default point to this state upon creating a new data set or attaching a data set variable to an existing data set.

ds.resetAtEnd

```
void ds.resetAtEnd()
```

This function resets the pointer past the last item in the data set.

ds.getCorrelationTime

```
long long ds.getCorrelationTime()
```

This function will return the Time value (in picoseconds) of the trigger position for this data set for use in correlating between multiple data sets.

ds.getCorrelationState

```
long long ds.getCorrelationState()
```

This function will return the State value of the trigger position for this data set for use in correlating between multiple data sets.

ds.getPosition

```
long long ds.getPosition()
```

This function will return the Time or State value of the current position for this data set depending on the current bias.

If the bias setting for *ds* is State, then the value found under the State column at the position of the pointer *ds* will be returned. If the bias setting for *ds* is Time, then the value found under the Time column at the position of the pointer *ds* will be returned. Time values are in picoseconds.

```
ds1.setTimeBias();
ds2.setTimeBias();
ds1.setPosition(ds2.getPosition());
```

ds.firstPosition

```
long long ds.firstPosition()
```

This function will return the Time or State value of the first position for this data set depending on the current bias.

If the bias setting for *ds* is State, then the value found under the State column at the position of the pointer *ds* will be returned. If the bias setting for *ds* is Time, then the value found under the Time column at the position of the pointer *ds* will be returned. Time values are in picoseconds.

```
ds1.setTimeBias();
ds2.setTimeBias();
ds1.setPosition(ds2.firstPosition());
```

ds.lastPosition

```
long long ds.lastPosition()
```

This function will return the Time or State value (in picoseconds) of the last position for this data set depending on the current bias.

If the bias setting for *ds* is State, then the value found under the State column at the position of the pointer *ds* will be returned. If the bias setting for *ds* is Time, then the value found under the Time column at the position of the pointer *ds* will be returned. Time values are in picoseconds.

ds.setPosition

```
int ds.setPosition(long long t)
```

The current position can be set with this function. The parameter *t* is interpreted according to the current bias as being Time or State information. This function has no effect on the label entries it contains. Returns an error code.

ds.getRunID

```
int ds.getRunID()
```

This function returns the run id of the data set. Data set ids are guaranteed to be the same if the data sets originated from the same run. This can be useful for group run situations to check whether two data sets originated from the same run.

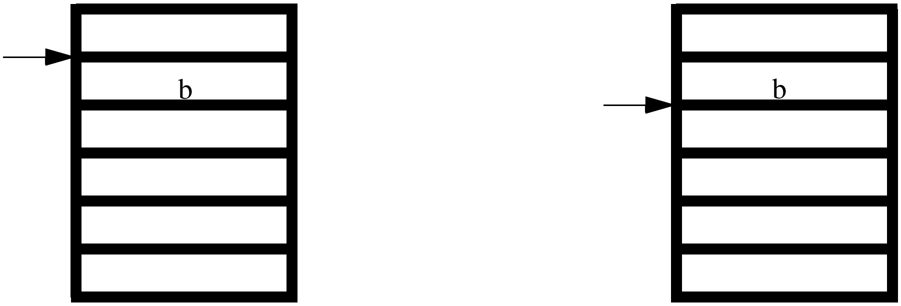
TDKDataSet Iteration Functions

The following functions are useful for stepping through the sampling information contained within data sets. Each data set variable maintains a pointer to a current position, which is shown in the figures below. Many of the functions either modify or refer to this pointer in some way.

ds.next

```
int ds.next(long long &t)
int ds.next()
```

This function sets the pointer to the next existing *x-axis* value within the data set and puts this value into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If it cannot return a valid *x-axis* position, then 0 is given as the return value. Otherwise the function returns 1.

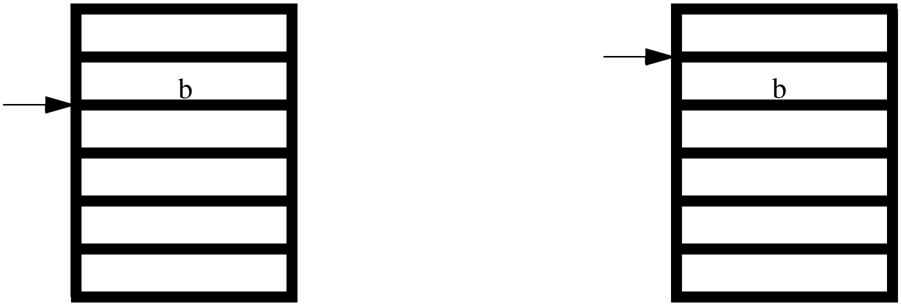


Before and after a call to next()

ds.prev

```
int ds.prev(long long &t)
int ds.prev()
```

This function sets the pointer to the previous existing *x-axis* value within the data set and puts this value into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If it cannot return a valid *x-axis* position, then 0 is given as the return value. Otherwise the function returns 1.



Before and after a call to prev()

ds.peekNext

```
int ds.peekNext(long long &t)
```

Without changing the value of the pointer, this function puts the value of the next time or state position into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If there is no valid next position the value 0 is returned, otherwise,1 is returned.

Before and after a call to peekNext()

ds.peekPrev

```
int ds.peekPrev(long long &t)
```

Without changing the value of the pointer, this function puts the value of the previous time or state position into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If there is no valid previous position the value 0 is returned, otherwise, 1 is returned.

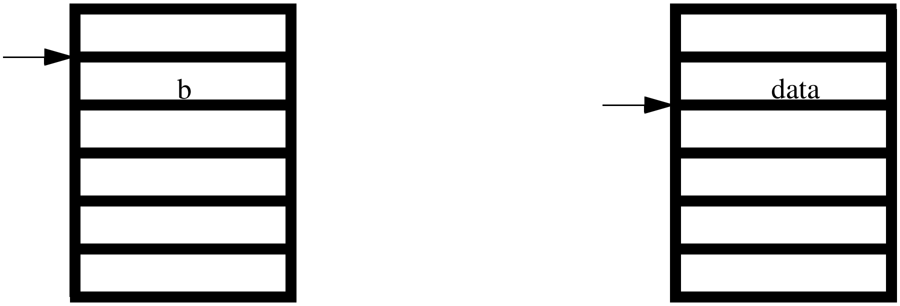
Before and after a call to peekPrev()

ds.replaceNext

```
int ds.replaceNext(long long &data)
```

Store the value of data in the position pointed to by the pointer, and then increment the pointer by one sample. This function is only valid for Time bias. If the bias setting for *ds* is Time, data is written to the Time sample entry. If the bias setting is State, this function will return 0 without any changes to the pointer. If the function cannot return a valid *x-axis* position, then 0 is given as the return value. Otherwise the function returns 1.

NOTE: Data sets created using the createTimePeriodic function cannot have their Time values modified since there are created with a constant time sampling period. This function will have no effect on data sets created as such.



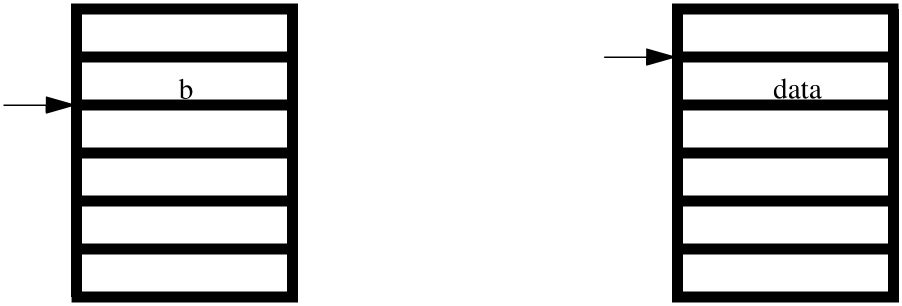
Before and after a call to replaceNext()

ds.replacePrev

```
int ds.replacePrev(long long &data)
```

Decrement the pointer by one sample, and then store the value of data in the position pointed to by the pointer. This function is only valid for Time bias. If the bias setting for *ds* is Time, data is written to the Time sample entry. If the bias setting is State, this function will return 0 without any changes to the pointer. If the function cannot return a valid *x-axis* position, then 0 is given as the return value. Otherwise the function returns 1.

NOTE: Data sets created using the createTimePeriodic function cannot have their Time values modified since there are created with a constant time sampling period. This function will have no effect on data sets created as such.



Before and after a call to replacePrev()

TDKDataSet Tutorials

An example will be presented to further demonstrate the concepts associated with accessing data in a data set. Following this example are several sample tutorials showing various data set tasks.

The table below shows a sample trace listing for a data set. This data set contains two label entries named "ADDR" and "DATA".

myData sample data set

State Number	ADDR	DATA	Time
0	0x1111	0x1212	1.0 ns
1	0x2222	0x2323	1.4 ns
2	0x3333	0x3434	1.8 ns
3	0x4444	0x4545	2.2 ns

Suppose, a data set variable *ds1* has been declared along with two label entry variables, *addr* and *data*. Also, assume that everything has been properly attached (data set to incoming data group, label entries to data set) and *ds1* points to the beginning of the trace. Next shown are various lines of code together with the results after execution of that code.

```
long long stateOrTimeValue;
ds1.reset( );
ds1.setTimeBias( );
ds1.next(stateOrTimeValue); // stateOrTimeValue = 1.0ns
ds1.next(stateOrTimeValue); // stateOrTimeValue = 1.4ns
ds1.setStateBias( );
ds1.next(stateOrTimeValue); // stateOrTimeValue = 2
ds1.next(stateOrTimeValue); // stateOrTimeValue = 3
```

Now, let's iterate through the data using the label entries variables. In a similar manner, the bias can be set on a label entry such that label entry iteration functions return either Time or State information depending on the bias setting. The choice is yours as to whether you wish to retrieve Time or State information through a data set iteration pointer or through a label entry iteration pointer. TDKLabelEntry iteration functions are discussed in the following section.

```
unsigned int dataValue;
TDKLabelEntry addr, data;
long long stateOrTimeValue;
addr.reset( );
addr.setTimeBias( );
addr.next( dataValue, stateOrTimeValue );
// dataValue = 0x1111, stateOrTimeValue = 1.0ns

addr.next( dataValue, stateOrTimeValue );
// dataValue = 0x2222, stateOrTimeValue = 1.4ns

data.reset( );
data.setStateBias( );
data.next( dataValue, stateOrTimeValue );
// dataValue = 0x1212, stateOrTimeValue = 0

data.next( dataValue, stateOrTimeValue );
// dataValue = 0x2323, stateOrTimeValue = 1
```

NOTE: It is not possible to retrieve ADDR or DATA data values through *ds1* the data set iteration pointer. It is only possible to retrieve Time or State information through the data set iteration pointer. However, it is possible to retrieve both ADDR and DATA data values together with Time and State information through a label entry iteration pointer such as *addr* or *data*.

Creating a new data set with a constant sampling frequency

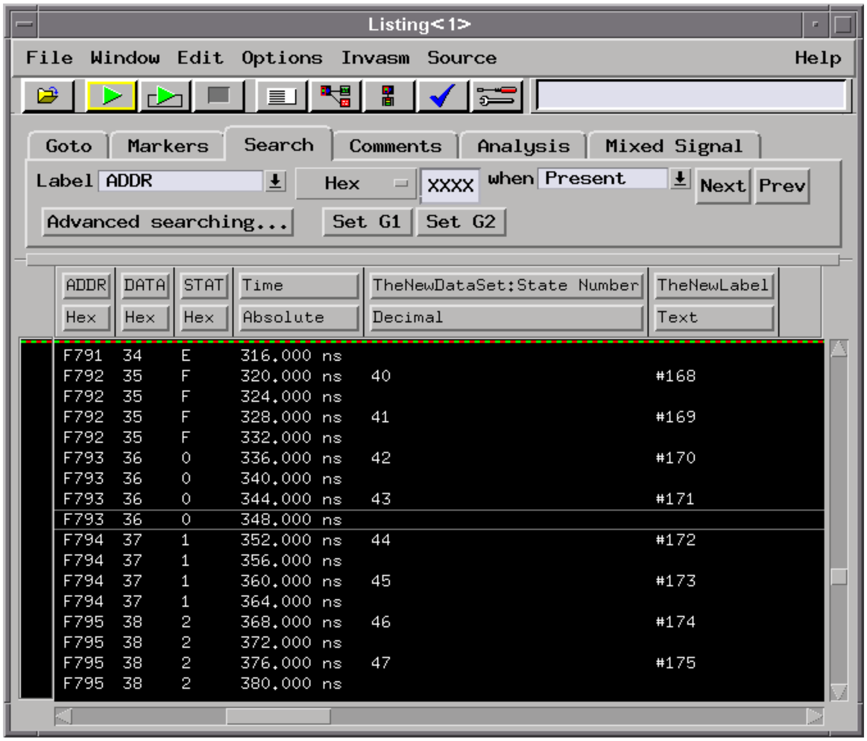
There are times when it would be convenient to add a new data set, and new label entries within that data set, where the new data set can have a different number of samples and a different constant sampling period than the incoming data set.

This example uses one input data set and creates a second data set with two times the acquisition period and half the number of samples using the createTimePeriodic() function.

1. Open the /logic/demo/ToolDevKit/sample9.___ configuration and the Tool Development Kit tool.
2. Select the Compile button, then Run.
3. View the modified data in the Lister.

a Use the horizontal scroll bar in the Lister window to view the new data set as shown in the following picture.

b You may view the original data by dropping a Lister at the output of the File In tool.



Label ADDR Hex XXXX when Present

```
ADDR  DATA  STAT  Time          TheNewDataSet:State Number   TheNewLabel
Hex   Hex   Hex   Absolute      Decimal                      Text
F791  34    E     316.000 ns
F792  35    F     320.000 ns    40                           #168
F792  35    F     324.000 ns
F792  35    F     328.000 ns    41                           #169
F792  35    F     332.000 ns
F793  36    0     336.000 ns    42                           #170
F793  36    0     340.000 ns
F793  36    0     344.000 ns    43                           #171
F793  36    0     348.000 ns
F794  37    1     352.000 ns    44                           #172
F794  37    1     356.000 ns
F794  37    1     360.000 ns    45                           #173
F794  37    1     364.000 ns
F795  38    2     368.000 ns    46                           #174
F795  38    2     372.000 ns
F795  38    2     376.000 ns    47                           #175
F795  38    2     380.000 ns
```

Results of creating a new Data Set with a constant sampling frequency

In this example, note that the setTimeCrossCorrelation() function is used so that downstream tools know that time correlated data is available.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample9.___ (Config file)
- sample.dat (Data file used by File In tool)

- sample9.c (Tool Development Kit program file)

sample9.c

```
//  File:  sample9.c
//  Purpose:  Create a new DataSet with a constant sampling
//            frequency


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  TDKLabelEntry le;
  TDKLabelEntry newLE;
  TDKDataSet ds;
  TDKDataSet newDS;
  char buf[20];
  String message;
  int i;
  long long time;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  // Assumed 4ns acquisition rate and 512 samples
  err = ds.attach( dg, “dataSet001” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the DATA label
  err = le.attach( ds, “DATA” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Create a new periodic time dataset with one label
  // New dataset has 8ns acquisition rate and 256 samples
  newDS.createTimePeriodic( dg, “TheNewDataSet”, 256u, 128u,
                             0, nanoSec(8.0) );

  // Create a new text labelentry in the new dataset
  newLE.createTextData( newDS, “TheNewLabel”, 10);

  // Fill in some values
  i = 0;

  while( newDS.next( time ) )
  {
    sprintf( buf, “#%d”, i++ );
    message = buf;
    newLE.replaceNext( message );
  }

  // Essential Step:  sets up necessary info so that downstream
  // tools know that time correlated data is available.
  dg.setTimeCrossCorrelation();
}
```

Creating a new data set with modifiable time stamps

There are times when it would be convenient to add a new data set, and new label entries within that data set, where the new data set has a different number of samples and sample frequency than the original data set and allows the time of each sample to be modified. By modifying the time stamps of the newly created data set, user-specified correlation with the incoming data is possible.

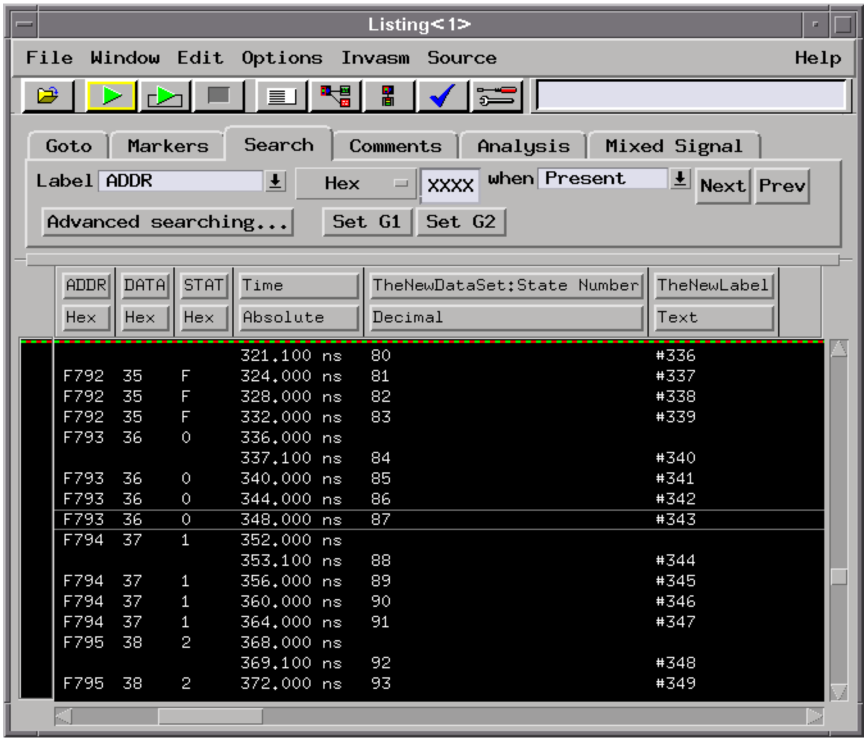
In this example the createTimeTags() function is used to create a new data set, and time tags are modified using the replaceNext() function.

1 Open the /logic/demo/ToolDevKit/sample10.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the modified data in the Lister.

  a You may view the original data by dropping a Lister at the output of the File In tool.



```
ADDR  DATA  STAT  Time         TheNewDataSet:State Number  TheNewLabel
Hex   Hex   Hex   Absolute     Decimal                     Text
                  321.100 ns   80                          #336
F792  35    F     324.000 ns   81                          #337
F792  35    F     328.000 ns   82                          #338
F792  35    F     332.000 ns   83                          #339
F793  36    0     336.000 ns
                  337.100 ns   84                          #340
F793  36    0     340.000 ns   85                          #341
F793  36    0     344.000 ns   86                          #342
F793  36    0     348.000 ns   87                          #343
F794  37    1     352.000 ns
                  353.100 ns   88                          #344
F794  37    1     356.000 ns   89                          #345
F794  37    1     360.000 ns   90                          #346
F794  37    1     364.000 ns   91                          #347
F795  38    2     368.000 ns
                  369.100 ns   92                          #348
F795  38    2     372.000 ns   93                          #349
```

Creation of a new Data Set with changeable time stamp

In this example, note that the setTimeCrossCorrelation() function is used so that downstream tools know that time correlated data is available.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample10.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample10.c (Tool Development Kit program file)

sample10.c

```
/*  File:  sample10.c
    Purpose:  Create a new DataSet with a different (and
              modifiable) sample period and a different
              number of samples than the original DataSet.

    When the createTimeTags() function is used to create
    the new DataSet, the time stamp (or time tag) on a
    state may be changed.
*/

void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  TDKLabelEntry le;
  TDKLabelEntry newLE;
  TDKDataSet ds;
  TDKDataSet newDS;
  char buf[20];
  String message;
  int i;
  long long time;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  // Assumed 4ns acquisition rate and 512 samples
  err = ds.attach( dg, “dataSet001” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }


  // Attach to the DATA label
  err = le.attach( ds, “DATA” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Create a new periodic time dataset with one label
  // New dataset has 4ns acquisition rate and 512 samples
  newDS.createTimeTags( dg, “TheNewDataSet”, 512u, 256u,
                        0, nanoSec(4.0) );

  // Create a new text labelentry in the new dataset
  newLE.createTextData( newDS, “TheNewLabel”, 10);

  // Fill in values for the new DataSet
  i = 0;
```

```
    while( newDS.peekNext( time ) )
    {
      sprintf( buf, "#%d", i);
      message = buf;
      newLE.replaceNext( message );

      // Every fourth state, add 1.1nS to the time
      if( (i % 4) == 0 )
      {
        time += nanoSec( 1.1 );
      }

      // Write out the potentially modified time
      newDS.replaceNext( time );

      i++;
    }

    // Essential Step: sets up necessary info so that downstream
    // tools know that time correlated data is available.
    dg.setTimeCrossCorrelation();
  }
```

Filtering data within the Tool Development Kit

It is possible to remove data from the incoming data stream through use of the filter() function. The pattern filter tool provides the ability to remove data from the incoming data stream based on simple pattern matching. When the pattern filter tool does not provide the needed functionality, a Tool Development Kit program can be written to provide the desired filtering.

This example will filter out every other state whose ADDR value matches the specified search value which is F794.

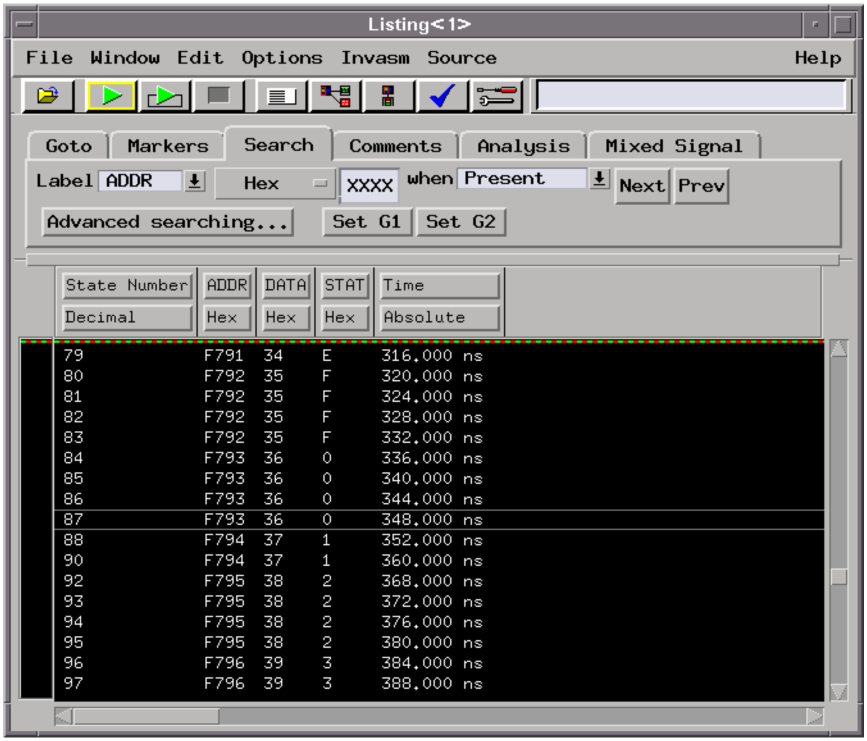
1 Open the /logic/demo/ToolDevKit/sample7.___ config and the Tool Development Kit tool.

2 Select the Compile, then Run.

3 View the modified data in the Lister.

   a Scroll to see that every other state whose ADDR value matches the search value 0xF794 has been filtered out.

   b You may view the original data by dropping a Lister at the output of the File In tool.


Listing<1>
File Window Edit Options Invasm Source Help
Goto Markers Search Comments Analysis Mixed Signal
Label ADDR Hex XXXX when Present Next Prev
Advanced searching... Set G1 Set G2

```
State Number  ADDR  DATA  STAT  Time
Decimal       Hex   Hex   Hex   Absolute
79            F791  34    E     316.000 ns
80            F792  35    F     320.000 ns
81            F792  35    F     324.000 ns
82            F792  35    F     328.000 ns
83            F792  35    F     332.000 ns
84            F793  36    0     336.000 ns
85            F793  36    0     340.000 ns
86            F793  36    0     344.000 ns
87            F793  36    0     348.000 ns
88            F794  37    1     352.000 ns
90            F794  37    1     360.000 ns
92            F795  38    2     368.000 ns
93            F795  38    2     372.000 ns
94            F795  38    2     376.000 ns
95            F795  38    2     380.000 ns
96            F796  39    3     384.000 ns
97            F796  39    3     388.000 ns
```

Results of filtering data: Note that state number 89 is no longer visible in the Listing display.

Note in this example that filtering requires a state number, and that the entire state is filtered, not just the label entry.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample7.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample7.c (Tool Development Kit program file)

sample7.c

```
/*  File:  sample7.c
    Purpose:  A program to demonstrate filtering
              data within the TDK.

              This program looks for an ADDR value
              of f794 hex, then filters out every
              other state with that ADDR value.
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry le;
  TDKDataSet ds;
  unsigned value;
  unsigned findValue = 0xf794;  // value to filter
  long long position;
  int toggle;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset and LabelEntry
  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  err = le.attach( ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // set the toggle
  toggle = 0;

  // The position variable will hold the state number.
  // Use the toggle to filter out every other occurrence
  // of the findValue

  while( ds.next( position ) && le.next( value ) )
  {
    if  ( value == findValue )
    {
      if( toggle )
      {
        ds.filter( position );
      }
      toggle = !toggle;
    }
  }


}
```

Working with TDKLabelEntries

TDKLabelEntries contain the data that we have acquired from the probe sources and upstream tools. The type of this data is the type of the label entry. This may be Integral (Unsigned int), Text (String) or Analog (Double). The width of both Integral and Text data may vary, depending on the application.

Each TDKLabelEntry maintains a table of sample values and symbol pairs. This table serves as a mapping from values to symbols (which are user-defined string quantities). These symbols serve as mnemonics for certain values that the user may see during a run of the Analyzer. The Tool Development Kit tool provides a means to lookup, but not to modify the symbols in the table.

Label entries are declared as follows:

```
TDKLabelEntry le; //declare a LabelEntry variable le
```

TDKLabelEntry Creation Functions

As with TDKDataSets, TDKLabelEntry variables must be instantiated with a call to a create() or attach() function.

le.create

```
int le.create( TDKDataSet ds, String name, TDKLabelEntry
orig )
```

This function makes *le* a copy of the TDKLabelEntry *orig*, which is passed as a parameter. This allows the tool to modify the incoming data, if necessary. Note that Bias and Position information are not copied over from *orig* to *le*.

```
TDKDataSet ds;
TDKLabelEntry le;
TDKLabelEntry orig;

ds.attach( dg );
orig.attach( ds, "MyOwnLabel" );
```

```
le.create( ds, "MyNewLabel", orig );
```

le.createTextData/IntegralData/AnalogData

```
int le.createTextData( TDKDataSet ds, String name, int
width )
```

```
int le.createIntegralData( TDKDataSet ds, String name,
int width )
```

```
int le.createAnalogData( TDKDataSet ds, String name,
double Offset, double FullScaleVolts )
```

These functions are used to create a brand new label entry for use by downstream tools, which the Tool Development Kit tool can enter information into using the replace functions.

Three types of label entries may be created, Integral, Analog, and Text, using the createIntegralData(), createAnalogData(), and createTextData() functions. See page 53 for more information on Integral, Analog, and Text data. In each of the different cases, a data set must be given. This data set is the one in which the label entry is to be created. A name is also required in each case. This will be the name of the label, as it will appear in the downstream tools. Finally, either a width parameter is required or a label entry parameter is required. The width tells how wide in bits, or characters the new label entry should be. Note that Text label entries can be arbitrarily wide, no matter what the parameter is. This only serves to tell the lister how wide to make the display column.

```
TDKDataSet ds;
TDKLabelEntry stringLab;
TDKLabelEntry intLab;

ds.attach( dg );
stringLab.createTextData(ds, "comments", 15);
intLab.createIntegralData(ds, "Data", 32);
```

le.attach

```
int le.attach( TDKDataSet ds, String name )
```

Similar to the attach function for TDKDataSets, this function makes *le* a variable for the given label entry called *name* which exists in the particular run of the Tool Development Kit Tool. A data set variable parameter is given to identify which data set the label entry belongs to. That data set is searched for the label entry called name, which must match exactly. The data set variable *ds* must be attached before calling this function. Returns an error code.

NOTE: Label entries that are attached may not be modified. In order to modify an incoming label entry, a copy must be made with the create() function.

```
TDKDataSet ds;
TDKLabelEntry le;
ds.attach( dg );
le.attach( ds, "MyOwnLabel" );
```

TDKLabelEntries Utility Functions

le.getName

```
String le.getName()
```

This function returns the name of the label entry.

```
io.print( le.getName() );
```

le.setName

```
void le.setName( String newName )
```

This function changes the name of the label entry to *newName*.

```
le.setName( "Something Different" );
```

le.isAttached

```
int le.isAttached()
```

This function returns true if the label entry has been successfully attach()ed or create()ed on this run.

le.getWidth

```
int le.getWidth()
```

This function returns the width in bits of the label entry.

```
io.print( le.getWidth() );
```

le.isTextData

```
int le.isTextData()
```

This function returns true if the label entry contains textual data.

```
if(le.isTextData())
{
    io.print( "text label" );
}
else if(le.isIntegralData())
{
    io.print( "integral data" );

}
```

le.isIntegralData

```
int le.isIntegralData()
```

This function returns true if the label entry contains integral data.

le.isAnalogData

```
int le.isAnalogData()
```

This function returns true if the label entry contains analog data.

TDKLabelEntry Time and State Functions

In a manner similar to TDKDataSets, TDKLabelEntries can access Time and State information. To access time stamps and state numbers, you must use the iterators (next( ), prev( ), replaceNext( ), etc) on the data set or label entry. Furthermore, TDKLabelEntries set the time or state bias with functions similar to TDKDataSets. See TDKDataSet Tutorials on page 90 for an example on how to do this using TDKDataSets and TDKLabelEntries.

Bias Setting

The following TDKLabelEntry functions are sensitive to the bias setting for a given label entry. Recall that the default bias setting is State. Bias settings for a label entry can be changed to either State or Time by using either le.setStateBias or le.setTimeBias.

- le.getPosition
- le.setPosition
- le.firstPosition
- le.lastPosition
- le.next
- le.prev
- le.peekNext
- le.peekPrev
- le.replaceNext
- le.replacePrev

For example, if the bias setting for a given label entry is time (le.setTimeBias explicitly called), then the function le.getPosition will return the value found under the Time column in the listing window. If the bias setting is State (default), then le.getPosition will return the value found under the State column in the listing window.

le.setTimeBias

```
int le.setTimeBias()
```

This function sets Timing bias for the label entry. Bias (state or timing) indicates the type of information the iteration functions will operate on. The default bias is State, since Timing information may not exist. Returns an error code.

le.setStateBias

```
int le.setStateBias()
```

This function sets State bias for the label entry. Bias (state or timing) indicates the type of information the iteration functions will operate on. The default bias is State, since Timing information may not exist. Returns an error code.

le.setPosition

```
int le.setPosition(long long s)
```

This function sets the pointer *le* to State number *s* or time *s*, depending on the current bias. If the bias setting is State, then *s* is interpretted as the State number value. If the bias setting is Time, then *s* is interpretted as the Time value in picoseconds. Because some states may have been filtered out, the given state *s* may not exist. In this case, the position will be set to the previous existing state. If no previous state exists, then the position will be set before the first state. Returns an error code.

```
le.setStateBias();
le2.setStateBias();
le.setPosition(le2.firstPosition());
```

le.getPosition

```
long long le.getPosition()
```

This function returns the State number or Time, depending on the bias, at the location of the pointer. If the bias setting for *le* is State, then the value found under the State column at the position of the pointer *le* will be returned. If the bias setting for *le* is Time, then the value found under the Time column at the position of the pointer *le* will be returned. Time values are in picoseconds.

le.firstPosition

```
long long le.firstPosition()
```

This function returns the State number or Time, depending on the bias, at the location of the first position. If the bias setting for *le* is State, then the value found under the State column at the position of the pointer *le* will be returned. If the bias setting for *le* is Time, then the value found under the Time column at the position of the pointer *le* will be returned. Time values are in picoseconds.

le.lastPosition

```
long long le.lastPosition()
```

This function returns the State number or Time, depending on the bias, at the location of the last position. If the bias setting for *le* is State, then the value found under the State column at the position of the pointer *le* will be returned. If the bias setting for *le* is Time, then the value found under the Time column at the position of the pointer *le* will be returned. Time values are in picoseconds.

TDKLabelEntry Iteration Functions

The following operations are used for iterating over the data contained within TDKLabelEntries. In order to keep track of the current position of iteration in the label entry, a pointer is maintained. Most of the functions below affect this pointer in some way. It is important to keep track of this pointer when writing Tool Development Kit code.

The diagrams show the state of the pointer both before (on the left) and after (on the right) the call to the function. The pointer is shown as pointing between samples. Unless the pointer is at the first or last position, there is always a next and a previous sample position. The descriptions of the following functions refer to the pointer position.

The functions that retrieve and replace data are overloaded to reflect the fact that label entries may contain several types of data (Integral - unsigned int, Text - String, Analog - double). In the future, other kinds of data may be added to the Agilent Technologies 16700 system. When this happens, the Tool Development Kit Tool, and all other tools, will be updated to support this new functionality. For example, additional overloaded functions will be added to address the new data types. The intention is to keep the old interface the same so that old tools will run, while at the same time, provide the newest level of functionality of the system to the programmer.

NOTE: All TDKLabelEntry iteration functions are overloaded to account for the underlying Integral, Analog, or Text data. It is important to use the correct overloaded function for the data type contained by the label entry. Use the functions le.isTextData(), le.isIntegralData(), or le.isAnalogData() to determine the data type for the label entry if you are not sure. Additionally, the View datagroup... option off the Tool Development Kit menu bar will show the data types for each label entry.

```
Here is a typical use of iteration:

/* ... declarations and attachments ... */
le.reset();
newLab.reset(); // make sure we are at the beginning
while( le.next( x ) )
    newLab.replaceNext( x );
```

The above code copies all the values from the label entry *le* to the label entry *newLab*. Note that x would be declared as String or unsigned depending on the kind of label entry we have.

le.reset

```
void le.reset()
```

This function resets the pointer before the first item in the label entry.

le.resetAtEnd

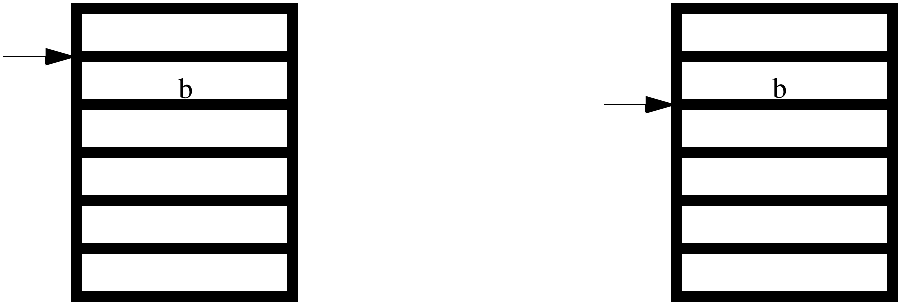
```
void le.resetAtEnd()
```

This function resets the pointer past the last item in the label entry.

le.next

```
int le.next(unsigned int &data) // Integral data
int le.next(String &data) // Text data
int le.next(double &data) // Analog data
int le.next(unsigned int &data, long long &pos) //
Integral data
int le.next(String &data, long long &pos) // Text data
int le.next(double &data, long long &pos) // Analog data
int le.next()
```

Fetch the data at the position of the pointer (b) into the variable *data* from the label entry and then increment the pointer by one sample. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds. This function may be called with no arguments to change the current position without fetching the data. Returns 1 if it is a valid sample, otherwise returns 0.



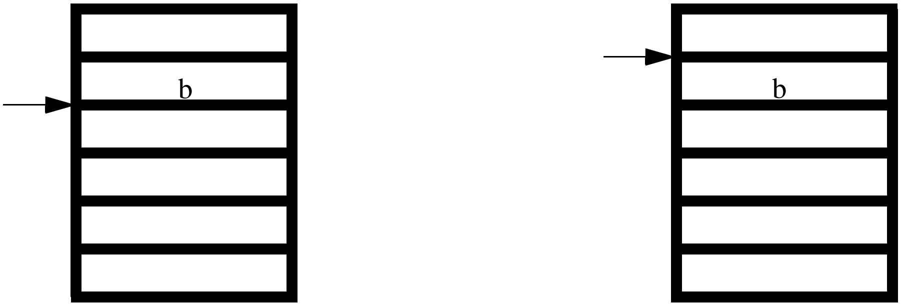
Before and after a call to next()

NOTE: Note, after the call to next, *pos* contains the same value as *le* before the call to next.

le.prev

```
int le.prev(unsigned int &data) // Integral data
int le.prev(String &data) // Text data
int le.prev(double &data) // Analog data
int le.prev(unsigned int &data, long long &pos) //
Integral data
int le.prev(String &data, long long &pos) // Text data
int le.prev(double &data, long long &pos) // Analog data
int le.prev()
```

Decrement the pointer by one sample and then fetch the data at the position of the pointer (b) into the variable *data* from the label entry. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds. This function may be called with no arguments to change the current position without fetching the data. Returns 1 if it is a valid sample, otherwise returns 0.

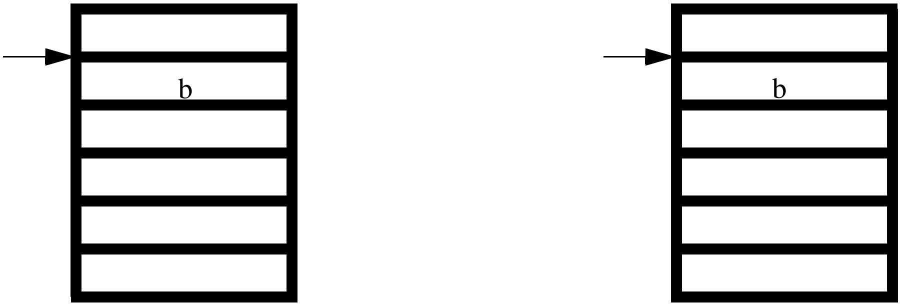


Before and after a call to prev()

le.peekNext

```
int le.peekNext(unsigned int &data) // Integral data
int le.peekNext(String &data) // Text data
int le.peekNext(double &data) // Analog data
int le.peekNext(unsigned int &data, long long &pos)//
Integral data
int le.peekNext(String &data, long long &pos) // Text
data
int le.peekNext(double &data, long long &pos) // Analog
data
```

Fetch the data from the label entry at the position of the pointer (c) into *data* but do not change the position of the pointer. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds. Returns 1 if it is a valid sample, otherwise returns 0.

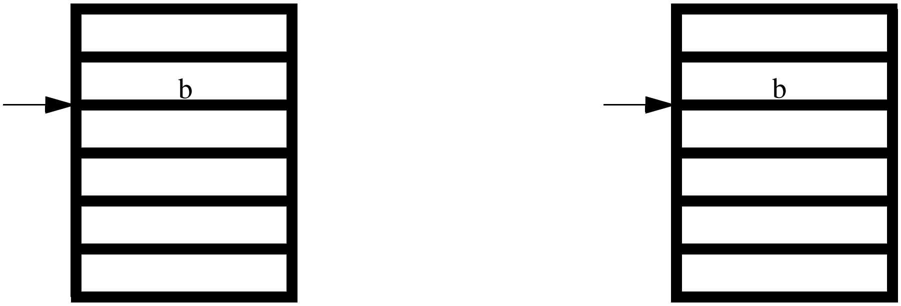


Before and after a call to peekNext()

le.peekPrev

```
int le.peekPrev(unsigned int &data) // Integral data
int le.peekPrev(String &data) // Text data
int le.peekPrev(double &data) // Analog data
int le.peekPrev(unsigned int &data, long long& pos) //
Integral data
int le.peekPrev(String &data, long long& pos) // Text
data
int le.peekPrev(double &data, long long& pos) // Analog
data
```

Fetch the data from the label entry at the position previous to the pointer (b) into *data* but do not change the position of the pointer. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds Returns 1 if it is a valid sample, otherwise returns 0.

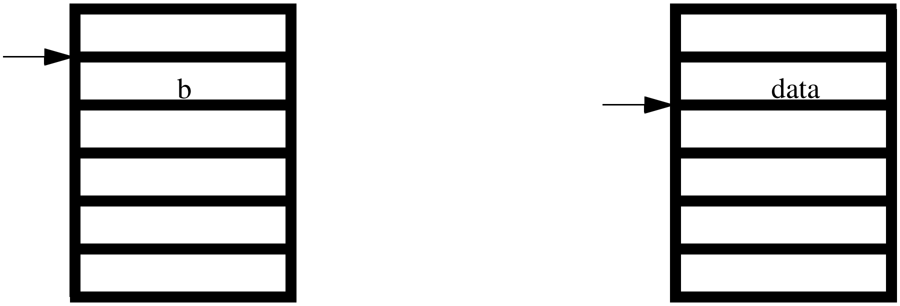


Before and after a call to peekPrev()

le.replaceNext

```
int le.replaceNext(unsigned int &data) // Integral data
int le.replaceNext(String &data) // Text data
int le.replaceNext(double &data) // Analog data
```

Store the value of *data* in the position pointed to by the pointer, and then increment the pointer by one sample. Returns 1 if it is a valid sample, otherwise returns 0.

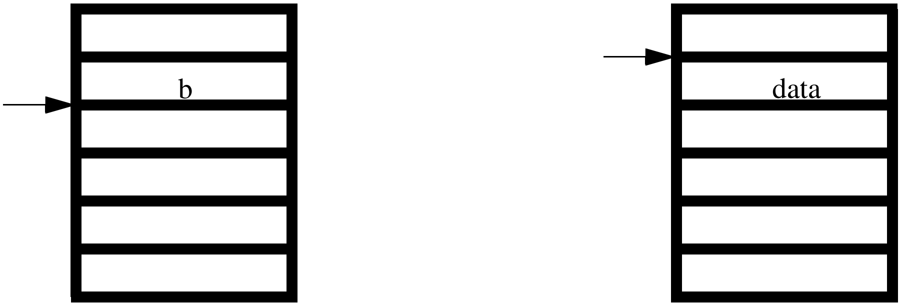


Before and after a call to replaceNext()

le.replacePrev

```
int le.replacePrev(unsigned int &data) // Integral data
int le.replacePrev(String &data) // Text data
int le.replacePrev(double &data) // Analog data
```

Decrement the pointer by one sample, and then store the value of *data* in the position pointed to by the pointer. Returns 1 if it is a valid sample, otherwise returns 0.



Before and after a call to replacePrev()

TDKLabelEntry Format Functions

le.formatSymbol

```
String le.formatSymbol(unsigned int val)
```

Look up *val* in the symbol table of the label entry and return its string symbol value. If it has none, the empty string is returned. See also the system routines formatXXX()s for converting sample values into different bases.

At this point conversion of a symbol to a value is not supported.

le.formatLine

```
String le.formatLine(unsigned int val)
```

This function returns the filename and line number information for a value *val* for use in source correlation. See also the system routines formatXXX()s for converting sample values into different bases.

le.formatHex/Oct/Dec/Bin/Twos

```
String le.formatHex(unsigned int val)
String le.formatOct(unsigned int val)
String le.formatDec(unsigned int val)
String le.formatBin(unsigned int val)
String le.formatTwos(unsigned int val)
```

This suite of functions is provided to convert the value *val* into different bases. These functions return string values that can be appended or included into other strings for output to down-stream tools. The returned string will have the same look as one that is formatted by the lister and other tools, for example. See also the label entry routine formatSymbol() for converting sample values to user-defined symbols.

TDKLabelEntry Highlighting Functions

Coloring and highlighting are mutually exclusive operations; you may do one of the two on any particular label entry.

le.setHighlight

```
int le.setHighlight(long long state)
```

This function displays the state *state* in a highlighted mode in a downstream Listing Tool. Returns an error code.

le.setColor

```
int le.setColor(long long state, int color)
```

This functions displays the state *state* in color *color* in a downstream Listing Tool. Returns an error code.

```
/* ... while iterating through the label entry le ...
*/
if( thisSampleIsFunny( le.getPosition() )
{
   le.setColor( le.getPosition(), 4 );
}
```

This example shows a potential use for coloring a label. The user's function thisSampleIsFunny() checks a position and returns true or false, then coloring is done.

TDKLabelEntry Searching Functions

The Searching routines below are inclusive, which means the search begins with the current sample.

Pattern Searching

The Pattern() search functions below use a value-mask scheme. A piece of data is compared to value. The bit pattern of mask is used to provide "don't care" bits to the comparison. A one bit is used to mean don't care in this case. Thus, if no masking is desired, a mask of 0 can be specified.

All the search routines return 1 for a successful search, and 0 otherwise.

le.searchRange.

```
int le.searchRange(long long& state, unsigned int lo,
unsigned int hi, int n)
```

This function finds the *nth* state from the current position whose data is between *lo* and *hi*, inclusive, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.searchNotRange.

```
int le.searchNotRange(long long& state, unsigned int lo,
unsigned int hi, int n)
```

This function finds the *nth* state from the current position whose data is not between *lo* and *hi*, inclusive, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.searchPattern.

```
int le.searchPattern(long long& state, unsigned int
value, unsigned int mask, int n)
```

This function finds the *nth* state from the current position whose data matches *value* and *mask*, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.searchNotPattern.

```
int le.searchNotPattern(long long& state, unsigned int
value, unsigned int mask, int n)
```

This function finds the *nth* state from the current position whose data does not match *value* and *mask*, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

String Searching

le.search.

```
int le.search(long long& state, String& value, int n,
eSearchMode mode)
```

This function finds the *nth* state from the current position whose data is the same as *value* if mode equals *Search_pattern*, and sets *state* to the position of the match. If mode equals *Search_notpattern*, the search finds the *nth* state from the current position whose data is not the same as *value* and sets *state* to the position of the non-match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found. eSearchMode is an enumerated type with the following values:

```
enum eSearchMode
{
   Search_pattern,
   Search_notpattern
}

long long state;
String value = "Bad data";
int n = 5000;
TDKLabelEntry::eSearchMode mode = TDKLabelEntry::Search_
pattern;

le.search(state, value, n, mode);
```

NOTE: The enumerated type eSearchMode is defined only for TDKLabelEntries and as such all variables and assignments to variables of that type must include the TDKLabelEntry scope operator "TDKLabelEntry::" as shown above.

This function will search from the current state up to 5000 states past the current state for data matching the text "Bad data" and will return the state position where "Bad data" is found in the variable state.

Highlighting

Coloring and highlighting are mutually exclusive operations; you may do one of the two on any particular label entry.

le.searchAndHighLightAllRange.

```
int le.searchAndHighLightAllRange(unsigned int lo,
unsigned int hi)
```

This function will display all states in a highlighted mode in the Listing Tool that are between *lo* and *hi*, inclusive. This function returns the number of states highlighted.

le.searchAndHighLightAllPattern.

```
int le.searchAndHighLightAllPattern(unsigned int value,
unsigned int mask)
```

This function will display all states in a highlighted mode in the Listing Tool that match *value* and *mask*. This function returns the number of states highlighted.

le.searchAndHighLightAllNotRange.

```
int le.searchAndHighLightAllNotRange(unsigned int lo,
unsigned int hi)
```

This function will display all states in a highlighted mode in the Listing Tool that are not between *lo* and *hi*, inclusive. This function returns the number of states highlighted.

le.searchAndHighLightAllNotPattern.

```
int le.searchAndHighLightAllNotPattern(unsigned int
value, unsigned int mask)
```

This function will display all states in a highlighted mode in the Listing

Tool that do not match *value* and *mask*. This function returns the number of states highlighted.

```
le.searchAndHighLightAllRange(1000, 1999);
```

The above call will highlight all states that have values in the [1000,2000) range.

Coloring

This suite of functions can be used to color all values that match a given criteria. The color parameter is a number between 0 and 7. Color is dependent upon the local settings of the colormap, which is user definable; therefore, it is impossible to determine how color will look on the end user's analyzer.

Coloring and Highlighting are mutually exclusive operations; you may do one of the two on any particular label entry.

le.searchAndColorAllRange.

```
int le.searchAndColorAllRange(int color, unsigned int
lo, unsigned int hi)
```

This function will display all states in color *color* in the Listing Tool that are between *lo* and *hi*, inclusive. This function returns the number of states colored.

le.searchAndColorAllPattern.

```
int le.searchAndColorAllPattern(int color, unsigned int
value, unsigned int mask)
```

This function will display all states in color *color* in the Listing Tool that match *value* and *mask*. This function returns the number of states colored.

le.searchAndColorAllNotRange.

```
int le.searchAndColorAllNotRange(int color, unsigned int
lo, unsigned int hi)
```

This function will display all states in color *color* in the Listing Tool that are not between *lo* and *hi*, inclusive. This function returns the number of states colored.

le.searchAndColorAllNotPattern.

```
int le.searchAndColorAllNotPattern(int color,
unsigned int value, unsigned int mask)
```

This function will display all states in color *color* in the Listing Tool that do not match *value* and *mask*. This function returns the number of states colored.

```
le.searchAndColorAllRange(4, 1000, 1999);
```

The above call will color all states that have values in the [1000,2000) range.

TDKLabelEntry Tutorials

The following pages provide several sample tutorials showing various label entry tasks.

Creating a New Numeric Data Label

In order to modify the incoming data, it is necessary to create a new label. The original data is read-only, but data in any newly created label entry may be changed.

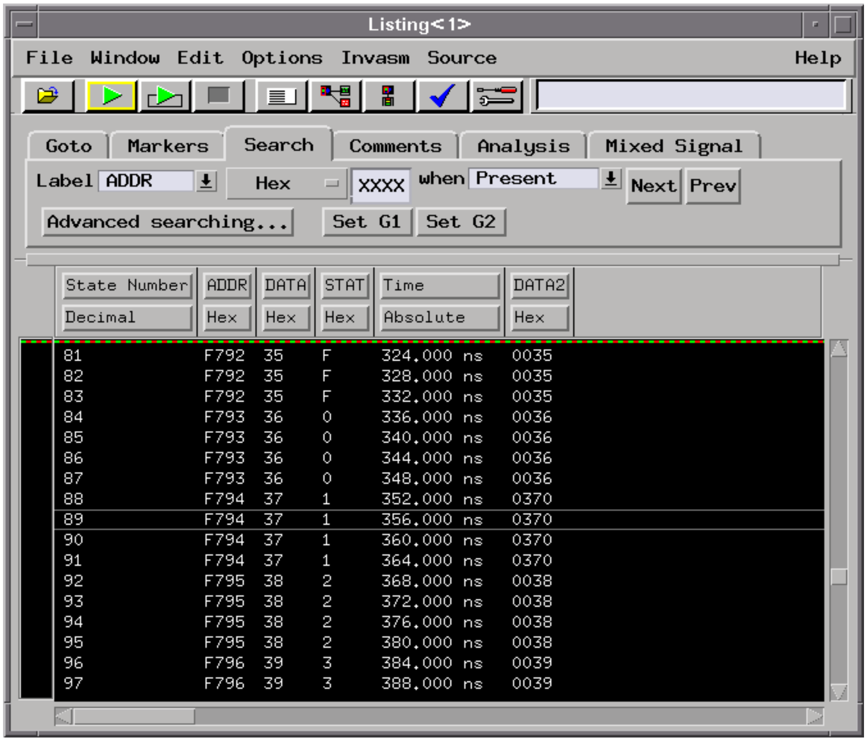
This example demonstrates creation of a numeric data label DATA2, and modification of data in the new label.

1 Open the /logic/demo/ToolDevKit/sample5.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the modified data in the Lister.

a A new data label called *Data2* is added to the listing. All the original *Data* values have been copied over to the new *Data2* label with the exception of wherever *ADDR* equals 0xF794 the new *Data2* value equals the original *Data* value multiplied by a constant of 16. This occurs at states 88 through 91.



Label ADDR Hex XXXX when Present

```
State Number  ADDR  DATA  STAT  Time          DATA2
Decimal       Hex   Hex   Hex   Absolute      Hex
81            F792  35    F     324.000 ns    0035
82            F792  35    F     328.000 ns    0035
83            F792  35    F     332.000 ns    0035
84            F793  36    0     336.000 ns    0036
85            F793  36    0     340.000 ns    0036
86            F793  36    0     344.000 ns    0036
87            F793  36    0     348.000 ns    0036
88            F794  37    1     352.000 ns    0370
89            F794  37    1     356.000 ns    0370
90            F794  37    1     360.000 ns    0370
91            F794  37    1     364.000 ns    0370
92            F795  38    2     368.000 ns    0038
93            F795  38    2     372.000 ns    0038
94            F795  38    2     376.000 ns    0038
95            F795  38    2     380.000 ns    0038
96            F796  39    3     384.000 ns    0039
97            F796  39    3     388.000 ns    0039
```

Creation of a new numeric data label

A label entry may contain data in one format, either numeric or text. This tutorial creates a label with a numeric format by using the createIntegralData() function.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample5.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample5.c (Tool Development Kit program file)

sample5.c

```
/*  File:  sample5.c
    Purpose:  A TDK program to create a new numeric data
              label, DATA2, then to copy the original value
              in the DATA label to the new label.


              If one particular ADDR value, f794 hex, is
              found, multiply the corresponding value
              in DATA2 by a constant, 16.
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry addrLE;
  TDKLabelEntry dataLE;
  TDKLabelEntry data2LE;
  TDKDataSet ds;

  unsigned addrValue;
  unsigned dataValue;
  unsigned data2Value;
  const int multConst = 16;     // multiplication constant
  unsigned searchValue = 0xf794; // ADDR value to find

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the “ADDR” label
  err = addrLE.attach( ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the DATA label
  err = dataLE.attach( ds, “DATA” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }


  // Create a new numeric data label, DATA2 which is
  // 16 bits wide in the same DataSet
  err = data2LE.createIntegralData( ds, “DATA2”, 16 );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Loop through the ADDR and DATA values and multiply
  // the DATA value by a constant when the ADDR search
  // value is found.  Otherwise, just copy the original
  // DATA value.

  while( addrLE.next( addrValue ) && dataLE.next( dataValue ) )
  {
    // Copy the dataValue
    data2Value = dataValue;

    // See if this is the desired ADDR value
    if (addrValue == searchValue)
    {
      data2Value *= multConst;
    }

    data2LE.replaceNext( data2Value );
  }
}
```

Creating a New Text Label

There are times when it is helpful to be able to write text comments or interpretations into a label entry. The Tool Development Kit allows you to create a text format label entry for this purpose.

In this example a text label entry is created, and when a specific search value is found in the STAT label, the word "Found" is written into the newly created text label.

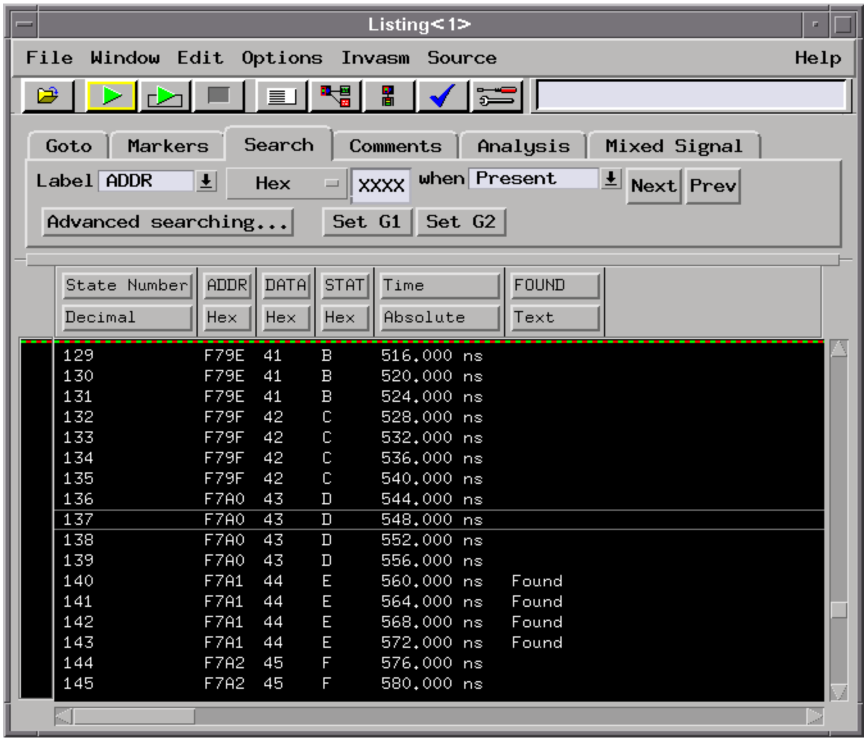
1 Open the /logic/demo/ToolDevKit/sample6.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the modified data in the Lister.

a Scroll to view "Found" in the "FOUND" label in states where the value in the STAT label is equal to 0E hex. States 76 through 79 are some of the states where this is true.

b You may view the original data by dropping a Lister at the output of the File In tool.



State Number Decimal	ADDR Hex	DATA Hex	STAT Hex	Time Absolute	FOUND Text
129	F79E	41	B	516.000 ns	
130	F79E	41	B	520.000 ns	
131	F79E	41	B	524.000 ns	
132	F79F	42	C	528.000 ns	
133	F79F	42	C	532.000 ns	
134	F79F	42	C	536.000 ns	
135	F79F	42	C	540.000 ns	
136	F7A0	43	D	544.000 ns	
137	F7A0	43	D	548.000 ns	
138	F7A0	43	D	552.000 ns	
139	F7A0	43	D	556.000 ns	
140	F7A1	44	E	560.000 ns	Found
141	F7A1	44	E	564.000 ns	Found
142	F7A1	44	E	568.000 ns	Found
143	F7A1	44	E	572.000 ns	Found
144	F7A2	45	F	576.000 ns	
145	F7A2	45	F	580.000 ns	

Creation of a new text label

In this example the createTextData() function was used to create a new text formatted label entry. The text was written out using the setPosition() and replaceNext() functions.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample6.___ (Config file)

- sample.dat (Data file used by File In tool)
- sample6.c (Tool Development Kit program file)

sample6.c

```
/*  File:  sample6.c
    Purpose:  A TDK program demonstrating how to create
              and enter text into a new text label

    Note that in this example we're looking for a value
    in the STAT label
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry statLE;
  TDKLabelEntry foundLE;
  TDKDataSet ds;

  unsigned statValue;
  String foundValue;
  long long position;
  // we will search for the value 0E hex in the STAT label
  unsigned searchValue = 0xE;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the STAT label
  err = statLE.attach( ds, "STAT" );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Create a new text LabelEntry for FOUND text 8 characters
  // wide
  err = foundLE.createTextData( ds, "FOUND", 8 );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // search for a STAT value equal to the searchValue
  // if found, write “Found” out to the new LabelEntry,
  // otherwise, just write a space

  while( statLE.next( statValue, position ) )
  {
    if( statValue == searchValue )
    {
      foundValue = “Found”;
    }
    else
    {
      foundValue = “ “;
    }

    // must set the foundLE iterator to the same position
    // as the statLE iterator, then write out the text
    foundLE.setPosition( position );
    foundLE.replaceNext( foundValue );
  }
}
```

Finding and Highlighting a Data Value

There are times when it is convenient to highlight a particular data value. Out of range values, error conditions, all occurrences of a particular value, etc., may be highlighted to be more easily visible.

This example program searches for one particular data value, then highlights that value in the Listing window when found.

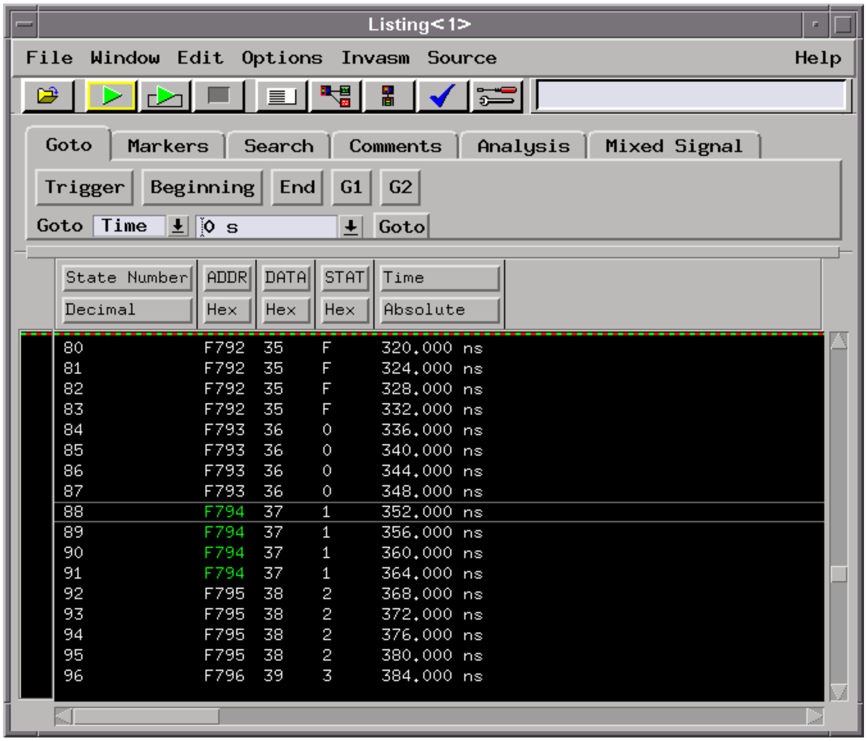
1 Open the /logic/demo/ToolDevKit/sample2.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the highlighted data in the Lister.

  a Open the Lister at the output of the Tool Development Kit by selecting the Lister icon and choosing Display. You may view the unmodified data by dropping a Lister at the output of the File In tool.

b Scroll to view the highlighted values which are found at states 88 through 91 in the Listing window.



```
State Number  ADDR  DATA  STAT  Time
Decimal       Hex   Hex   Hex   Absolute
80            F792  35    F     320.000 ns
81            F792  35    F     324.000 ns
82            F792  35    F     328.000 ns
83            F792  35    F     332.000 ns
84            F793  36    0     336.000 ns
85            F793  36    0     340.000 ns
86            F793  36    0     344.000 ns
87            F793  36    0     348.000 ns
88            F794  37    1     352.000 ns
89            F794  37    1     356.000 ns
90            F794  37    1     360.000 ns
91            F794  37    1     364.000 ns
92            F795  38    2     368.000 ns
93            F795  38    2     372.000 ns
94            F795  38    2     376.000 ns
95            F795  38    2     380.000 ns
96            F796  39    3     384.000 ns
```

Results of the Highlighting sample

Note that the setHighlight() function requires a state number. In this example, the state number is held in the “position” variable, and is returned by the ds.next(position) function call. Also note that the highlighting only affects data displayed in the Lister that has a Tool Development Kit tool in its data flow.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample2.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample2.c (Tool Development Kit program file)

sample2.c

```
/*  File:  sample2.c
    Purpose:  A TDK program to search for a particular
              value, then highlight that ADDR label

              The search value used is f794 hex, and
              we are looking for that value in the ADDR
              label.
*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry le;
  TDKDataSet ds;
  unsigned value;
  unsigned findValue = 0xf794;  // value to search for
  long long position;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the “ADDR” label which is found in the dataset
  err = le.attach( ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // the position variable will hold the state number
  while( ds.next(position) && le.next(value) )
  {
    if( value == findValue )
    {
      io.printf( “Pattern found at %d”, position );
      le.setHighlight(position);
    }
  }


}
```

Finding and Coloring a Data Value

Coloring data is another way in addition to highlighting to make certain data values more easily visible.

This example demonstrates searching for a particular data value and coloring the value when found using the setColor() function.

1 Open the /logic/demo/ToolDevKit/sample3.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the modified data in the Listing.

   a You may view the unmodified data by dropping a second Listing at the output of the File In tool.

   b Scroll to view the colorized values.

Output of this program is similar to the listing shown for the Highlighting a Data Value example.

Eight colors are available which depend on the current color palette of the analyzer as set up by the user:

0 White

1 White

2 Scarlet

3 Pumpkin

4 Yellow

5 Lime

6 Turquoise

7 Lavender

The setColor() function also requires the state number. This example shows how to retrieve the state number through use of the next(value, position) function call. Note that the coloring only affects data displayed in the Lister that has the Tool Development Kit tool in its data path.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample3.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample3.c (Tool Development Kit program file)

sample3.c

```
/*  File:  sample3.c
    Purpose:  A TDK program to search for a particular
              value, then set the color for that LabelEntry
              at that position.

              Our search value is f794 hex, and we are
              looking for that value in the ADDR label.
*/


// enumerated type to make the use of colors more clear:
// The numbers 1 - 8 are assigned to colors.  The "invalid"
// assignment is a place-holder, since enum{} will assign
// 0 to the first variable name.  Using 0 in the setColor()
// function results in the default color white.

enum { white, white2, scarlet, pumpkin, yellow, lime,
turquoise, lavender };

void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry le;
  TDKDataSet ds;
  unsigned value;
  unsigned findValue = 0xf794;  // value to search for
  long long position;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the “ADDR” label which is found in the dataset
  err = le.attach( ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // the position variable will hold the state number
  // this sample shows another way of retrieving the posiiton

  while( le.next( value, position ) )
  {
    if( value == findValue )
    {
      io.printf( “Pattern found at %lld”, position );
      le.setColor( position, scarlet );
    }
  }
}
```

Performing a Pattern-match Search

One important task that you may accomplish with the Tool Development Kit is searching for data which meets certain specifications. Searches may be performed by matching patterns or by finding data within a range of values.

This example demonstrates searching for a pattern using a value and a mask, then printing the location when found.

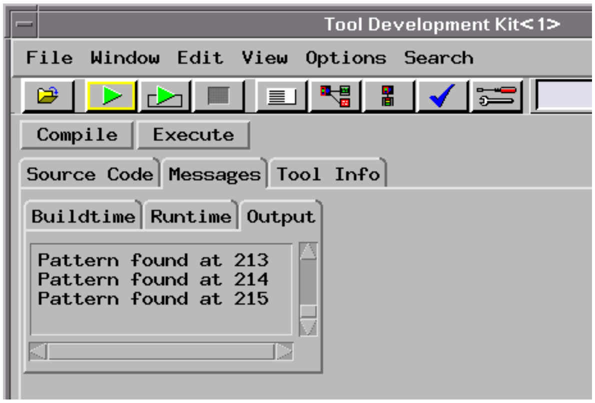
1 Open the /logic/demo/ToolDevKit/sample4.___ configuration and the Tool Development Kit tool.

2 Select the Compile button.

3 Select the Run button.

4 You may view the data in the Lister. The program outputs the states where the pattern for the *ADDR* label is equal to value 0x0023 with don't care mask 0xFFD0. Those states are outputted

in the Tool Development Kit Output window through the use of the io.printf function. The ADDR pattern is first found at state number -108 and last found at state 215.


Tool Development Kit<1>
File Window Edit View Options Search
Compile Execute
Source Code Messages Tool Info
Buildtime Runtime Output

```
Pattern found at 213
Pattern found at 214
Pattern found at 215
```

Results of the Pattern-Match sample

This example demonstrates the use of the Tool Development Kit's built in searching capabilities. Data searching can be performed on pattern values (using the searchPattern() function) or range values (using the searchRange() function). Note that when using the mask, a bit value of 0 indicates that the mask bit must equal the corresponding value bit, and a mask bit value of 1 indicates a don't care for the corresponding value bit.

Files Used:

From the /logic/demo/ToolDevKit/ directory.

- sample4.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample4.c (Tool Development Kit program file)

sample4.c

```
/*  File:  sample4.c
    Purpose:  A TDK program to perform a pattern-match search
              using a value and a mask, and display the state
              numbers at which the pattern is found.

We are using value = 0023 hex
             mask  = ffd0 hex
```

```
                bit 15                   bit 0
                     \                   /
In binary, the mask = 1111 1111 1101 0000
      and the value = 0000 0000 0010 0011

The 1's in the mask represent don't care positions.  The only
positions we will match in the value are the positions
in the mask which are equal to 0, in this case we will match
bits 0, 1, 2, 3, and 5.

So, if we use an "x" to represent "don't care", when we do
our pattern match we will be looking for:
                            xxxx xxxx xx1x 0011


*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry le;
  TDKDataSet ds;
  unsigned value;
  unsigned mask;
  long long position;

  // variable for keeping track of error codes
  int err;

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the "ADDR" label which is found in the dataset
  err = le.attach( ds, "ADDR" );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // This search asumes that the ADDR label is 16 bits.
  // In this search we are looking for the pattern
  // xxxx xxxx xx1x 0011 binary
  // (here, x means don't care)

  value = 0x0023;  // 0x here indicates a hex value
  mask = 0xffd0;

  // The position variable will hold the state number.
  // The state number will be printed if the search
  // pattern is found.
  while( le.searchPattern( position, value, mask, 1 ) )
  {
    io.printf( “Pattern found at %lld”, position );
  }
}
```

Working with TDKCorrelators and TDKCorrelatorValues

A TDKCorrelator is a data type that allows iteration over a set of TDKLabelEntries that belong to different data sets and are correlated in Time or State. This allows for multiple incoming TDKDataSets to be "matched-up" to one another. The correlation can be done in terms of State or Time (the default) information, through the setBias functions.

The set of TDKLabelEntries involved in the correlation is encapsulated into another data type called a TDKCorrelatorValue (CV). CVs are used during iteration as a way of returning all the information about all the label entries at that point in the correlation.

Correlators are declared as follows:

```
//declare a TDKCorrelator variable c
TDKCorrelator c;
```

TDKCorrelatorValues are used in the correlation of data in conjunction with a correlator. CVs contain all the information about all the label entries involved in the Correlation. A correlator value has three states, Valid, Held, and Changed. These states can be checked with the functions isValid(), isHeld() and isChanged().

TDKCorrelatorValues are declared as follows:

```
//declare a TDKCorrelatorValue variable cv
TDKCorrelatorValue cv;
```

TDKCorrelator Functions

As with data sets and label entry variables, TDKCorrelators must be instantiated with a function call. TDKCorrelators are instantiated with the initialize() function.

Correlators require a "reference TDKDataSet" be given (through the initialize() function) in order to set up a global state/timing reference for all the other data sets involved in the correlator.

c.initialize

```
int c.initialize( TDKDataSet referenceDataSet)
int c.initialize( TDKDataSet referenceDataSet,
TDKLabelEntryList leList )
```

The initialize functions prepare the data sets to be correlated. In its second form, an array of label entries is given. These are the label entries that are to be correlated, otherwise all label entries are involved in the correlation. *referenceDataSet* is the data set which is used for position information in the correlation process. Note that leList may contain label entries from various data sets including the referenced data set. Returns an error code.

```
TDKCorrelator c;
TDKCorrelatorValue cv;
TDKDataSet refDS;
TKDLabelEntryList labels;

int i;
unsigned data;
/* do attach() on refDS, and labels[0]..labels[n] */
c.initialize(refDS, labels);
c.setTimeBias();
while( c.next(cv) )
{
   io.printf( "time = %lld", c.getPosition() );
   for(i = 0; i < labels.length(); i++)
   {
     if(cv.isChanged(labels[i]))
     {
        cv.getData(labels[i], data);
        io.printf( "%s = %u", labels[i].getName(),data );
     }
   }
}
```

The above example shows a correlated iteration that prints the time at each iteration and then shows the values of all the labels that changed at this point in the iteration.

Bias Setting

The following TDKCorrelator functions are sensitive to the bias setting for a given label entry. Recall that the default bias setting is State. Bias settings for a correlator can be changed to either State or Time by using either c.setStateBias or c.setTimeBias.

- c.getPosition
- c.setPosition
- c.firstPosition
- c.lastPosition
- c.next
- c.prev
- c.peekNext
- c.peekPrev

For example, if the bias setting for a given correlator is time (c.setTimeBias explicitly called), then the function c.getPosition will return the value found under the Time column in the listing window. If the bias setting is State (default), then c.getPosition will return the value found under the State column in the listing window.

c.setTimeBias

```
int c.setTimeBias()
```

This function sets Timing bias (the default) for the correlator. Bias (State or Timing) indicates the type of information the rest of these functions will operate on. This function must be called before the call to initialize(). Returns an error code.

c.setStateBias

```
int c.setStateBias()
```

This function sets State bias for the correlator. Bias (State or Timing) indicates the type of information the rest of these functions will operate on. This function must be called before the call to initialize(). Returns an error code.

c.getPosition

```
long long c.getPosition()
```

This function will return the Time or State value of the current position for the correlator depending on the current bias. If the bias setting for *c* is State, then the value found under the State column at the position of the pointer *c* will be returned. If the bias setting for *c* is Time, then the value found under the Time column at the position of the pointer *c* will be returned. Time values are in picoseconds.

c.setPosition

```
int c.setPosition(long long position)
int c.setPosition(TDKDataSet ds, long long position)
```

The setPosition() functions reset the position of the pointer to *position*. In the first form, the position is taken from the reference data set, while in the second form, the position is taken for the TDKDataSet *ds*, which may be any data set involved in the correlation. Returns an error code.

c.reset

```
int c.reset()
```

This function resets the position of the pointer to the first position. Returns an error code.

c.resetAtEnd

```
int c.resetAtEnd()
```

This function resets the position of the pointer to the last position. Returns an error code.

c.firstPosition

```
long long c.firstPosition()
```

This function will return the Time or State value of the first position for this correlator depending on the current bias. If the bias setting for *c* is State, then the value found under the State column at the position of the pointer *c* will be returned. If the bias setting for *c* is Time, then the value found under the Time column at the position of the pointer *c* will be returned. Time values are in picoseconds.

c.lastPosition

```
long long c.lastPosition()
```

This function will return the Time or State value of the last position for this correlator depending on the current bias. If the bias setting for *c* is State, then the value found under the State column at the position of the pointer *c* will be returned. If the bias setting for *c* is Time, then the value found under the Time column at the position of the pointer *c* will be returned. Time values are in picoseconds.

c.next

```
int c.next( TDKCorrelatorValue& cv)
```

This function gets the next TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It advances the pointer to the next sample.

c.peekNext

```
int c.peekNext( TDKCorrelatorValue& cv)
```

This function gets the next TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It does not move the pointer.

c.prev

```
int c.prev(TDKCorrelatorValue& cv)
```

This function gets the previous TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It moves the pointer back to the previous sample.

c.peekPrev

```
int c.peekPrev(TDKCorrelatorValue& cv)
```

This function gets the previous TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It does not change the position of the pointer.

TDKCorrelatorValue Functions

TDKCorrelatorValues are only instantiated through a call to one of the iterator functions of a TDKCorrelator. Once that has been done, the following functions can be used to get the information about the TDKCorrelatorValue that was returned. For the following functions *cv* is of type TDKCorrelatorValue.

cv.isValid

```
int cv.isValid(TDKLabelEntry le)
```

This function returns true (1) if the TDKCorrelatorValue *cv* contains valid information for the given label entry otherwise returns false (0).

cv.isHeld

```
int cv.isHeld(TDKLabelEntry le)
```

This function returns true (1) if the TDKCorrelatorValue *cv* contains held information for the given label entry otherwise returns false (0). Often times held data values are discarded.

cv.isChanged

```
int cv.isChanged(TDKLabelEntry le)
```

This function returns true if the TDKCorrelatorValue *cv* contains changed information for the given label entry. The actual data found at the label entry position pointed to by *cv* is compared to the previous *cv* label entry data value. If these two values are the same, then this function returns false (0) otherwise it returns true (1).

cv.getData

```
int cv.getData(TDKLabelEntry le, unsigned int &d) //
Integral data
```

```
int cv.getData(TDKLabelEntry le, String &d) // Text data
int cv.getData(TDKLabelEntry le, double &d) // Analog
data
```

This function returns true (1) if the TDKCorrelatorValue *cv* contains valid information for label entry *le* otherwise it returns false (0). The value is stored in *d*.

cv.getState

```
long long cv.getState(TDKLabelEntry le)
```

This function returns the State number of the label entry *le* at the current position of the TDKCorrelatorValue *cv*.

TDKCorrelator and TDKCorrelatorValue Tutorials

The following example is used to show various TDKCorrelator and TDKCorrelatorValue concepts.

Suppose we wish to correlate in time two data sets, *Data Set 1* and *Data Set 2*. *Data Set 1* contains *labelentry1* and *Data Set 2* contains *labelentry2*. A sample trace listing and TDKCorrelatorValue operations will be shown to illustrate the concepts. Assume there is a TDKCorrelator variable *c* declared that has been set to Time bias and is reset to point to the first sample in the Trace listing below. Furthermore, assume there is a TDKCorrelatorValue variable *cv*. It may be helpful to think of *c* as a pointer to a row of information in the listing and *cv* as the actual row of data pointed to by *c*. Thus, executing

```
c.next( cv )
```

increments the pointer *c* one more row down into the listing, while *cv* contains the data found across that row prior to incrementing the pointer. The operations shown in the following table show the results from executing the following statements:

```
cv.isValid ( labelentry1 );
cv.isHeld ( labelentry1 );
cv.isChanged ( labelentry1 );
cv.getData( labelentry1, data );   // data is of type
```

```
unsigned int - Integral data assumed
```

Similarly, the same operation are performed for labelentry2. Assume c.next(cv) is executed for each sample in the trace listing such that *cv* contains the row of data of interest.

Data Set 1 labelentry1	Data Set 2 labelentry2	Time	cv operations on labelentry1	cv operations on labelentry2

0x1234	No data	1.1 ns	isValid = True
			isHeld = False
			isChanged = True
			getData = 0x1234
				isValid = False
				isHeld = False
				isChanged = False
				getData = -----

0x5678	No data	1.2 ns	isValid = True
			isHeld = False
			isChanged = True
			getData = 0x5678
				isValid = False
				isHeld = False
				isChanged = False
				getData = -----

Not Valid	0x1111	1.3 ns	isValid = True
			isHeld = True
			isChanged = True
			getData = 0x5678
				isValid = True
				isHeld = False
				isChanged = True
				getData = 0x1111

Not Valid	0x1111	1.4 ns	isValid = True
			isHeld = True
			isChanged = False
			getData = 0x5678
				isValid = True
				isHeld = False
				isChanged = False
				getData = 0x1111

0x9ABC	Not Valid	1.5 ns	isValid = True
			isHeld = False
			isChanged = True
			getData = 0x9ABC
				isValid = True
				isHeld = True
				isChanged = True
				getData = 0x1111

0x1234	0x2222	1.6 ns	isValid = True
			isHeld = False
			isChanged = True
			getData = 0x1234
				isValid = True
				isHeld = False
				isChanged = True
				getData = 0x2222

0x1234	No data	1.7 ns	isValid = True
			isHeld = False
			isChanged = False
			getData = 0x1234
				isValid = False
				isHeld = False
				isChanged = False
				getData = -----

Some observations to make from this sample listing:

- Whenever isValid returns False, do not call getData since it will contain a random value (a.k.a. garbage).

- Whenever isValid returns True, you probably still want to check if the result of isHeld is True. Often these values are discarded since they are not real samples for that data set.

Time Correlated Data Access

In a situation where you have two or more sets of incoming data, you may want to access the data in a time correlated fashion -- that is, access the data based on the order that it was sampled in time, regardless of which data source it came from. The Tool Development Kit provides a method for accessing data in a time correlated fashion-- the TDKCorrelator.

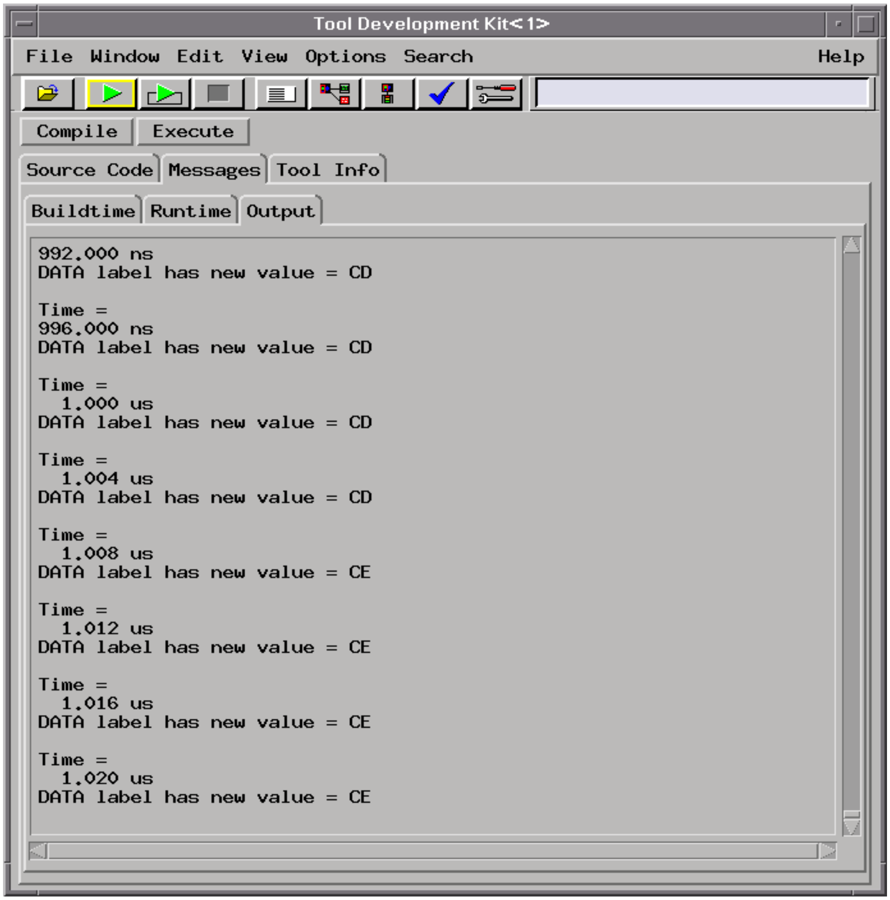
This example demonstrates the use of the TDKCorrelator using two sources of input data provided through the File In tool. The program iterates through values in the two data sets and writes data values to the output window in order of their occurrence in time.

1 Open the /logic/demo/ToolDevKit/sample8.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the modified data in the Lister.

a You may view the original data by dropping a Lister at the output of the File In tool.


Tool Development Kit<1>
File Window Edit View Options Search Help
Compile Execute
Source Code Messages Tool Info
Buildtime Runtime Output

```
992.000 ns
DATA label has new value = CD

Time =
996.000 ns
DATA label has new value = CD

Time =
  1.000 us
DATA label has new value = CD

Time =
  1.004 us
DATA label has new value = CD

Time =
  1.008 us
DATA label has new value = CE

Time =
  1.012 us
DATA label has new value = CE

Time =
  1.016 us
DATA label has new value = CE

Time =
  1.020 us
DATA label has new value = CE
```

Output of the time correlator example

This example introduces two new variable types: TDKCorrelator and TDKCorrelatorValue. In addition, another method of declaring label entries is demonstrated in this example: label entries are declared in an array.

The TDKCorrelator is a variable type that allows iteration over a set of label entries that are correlated in time or state. This allows for multiple incoming data sets to be "matched-up" to one another. The correlation can be done in terms of time (the default) or state information through use of the setTimeBias() or setStateBias() functions. In this example, time correlation is used.

The set of label entries involved in the correlation is encapsulated into another variable data type called a TDKCorrelatorValue. The TDKCorrelatorValues are used during iteration to return all the information about all the label entries at that point in the correlation. In this example, all label entries are correlated, and the *cv* variable contains time information for each of the label entries. The isChanged() function is used to determine which of the label entries has changed first.

Files Used:

From the /logic/demo/ToolDevKit/ directory.

- sample8.___ (Config file)
- sample8.dat (Second data file used by File In tool)
- sample8.c (Tool Development Kit program file)

sample8.c

```
// File:  sample8.c
// Purpose:  A TDK program to demonstrate the use of time
//           correlation with 2 sources of input data


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  // In this example, the LabelEntries are declared
  // in an array
  TDKLabelEntry le[2];
  TDKDataSet m1;
  TDKDataSet m2;
  unsigned value;
  long long position;

  // The Correlator variable type allows iteration
  // over a set of LabelEntries which are correlated
  // in time (or state if state bias is declared)
  TDKCorrelator c;

  // Declare the correlator variable which will
  // contain information about all of the LabelEntries
  // at a particular point in the correlation
  TDKCorrelatorValue cv;
```

```
// variable for keeping track of error codes
int err;

// Attach to the incoming dataset
err = m1.attach( dg, “dataSet001” );

if( err )
{
  // if we received an error, then print the message and exit
  io.printError( err );
  return;
}

err = m2.attach(dg, “dataSet002” );

if( err )
{
  // if we received an error, then print the message and exit
  io.printError( err );
  return;
}

// Attach to the DATA label
err = le[0].attach( m1, “DATA” );

if( err )
{
  // if we received an error, then print the message and exit
  io.printError( err );
  return;
}

// Attach to the ADDR2 label
err = le[1].attach( m2, “ADDR2” );

if( err )
{
  // if we received an error, then print the message and exit
  io.printError( err );
  return;
}

// Initialize the Correlator
err = c.initialize( m1 );

if( err )
{
  // if we received an error, then print the message and exit
  io.printError( err );
  return;
}

// Correlation will be based on
// time rather than state
c.setTimeBias();
```

```
  // Iterate over the selected labels and
  // see which label has a changed data value
  while( c.next( cv ) )
  {
    // print the time
    io.printf( “\nTime = “ );
    io.printf(timeToString( c.getPosition()) );

    // Check to see if the DATA label has new data
    if ( cv.isChanged( le[0] ) )
    {
      cv.getData( le[0], value );
      io.printf( “DATA label has new value = %0X”, value );
    }
    // Check to see if the ADDR2 label has new data
    if ( cv.isChanged( le[1] ) )
    {
      cv.getData( le[1], value );
      io.printf( “ADDR2 label has new value = %0X”, value );
    }
  }
}
```

TDK Miscellaneous Tasks

The following three tutorials show how to perform reading and writing of data to and from a file on the disk along with creating an installable tool.

File I/O: Reading Data from a File on the Disk

At times you may want to read data from a file on the disk. File I/O is accomplished using standard C functions.

This example demonstrates reading from a file on the disk using the fscanf() function.

1 Open the /logic/demo/ToolDevKit/sample13.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the modified data in the Lister. You may need to scroll the Listing window to view the new data set. If desired, you can view the file "/logic/demo/ToolDevKit/data.txt" through the Tool Development Kit editor by opening the file as a source file.

In this example note that the standard C language fscanf() formatting procedures are used.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- Sample13.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample13.c (Tool Development Kit program file)
- data.txt (Data file read during file I/O operation)

sample13.c

```
/*  File:  sample13.c
    Purpose:  Demonstrate file I/O by reading from a file on
              the disk.

    Create a new DataSet from data in a text file,
    data.txt, which contains 512 samples of 16 bit data
*/

void execute(TDKDataGroup &dg, TDKBaseIO &io)
{
  // Define variable name for the text data input file
  const char *dat = "/logic/demo/ToolDevKit/data.txt";

  // Declare other program variables

  TDKDataSet ds;
  TDKLabelEntry le;
  TDKDataSet newds;
  TDKLabelEntry dataLE;
  int count = 0;
  unsigned readValue;
  long long time = 0;

  // variable for keeping track of error codes
  int err;

  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  ds.setTimeBias();

  // Creating the new DataSet with the createTimeTags()
  // function will allow us to change the time stamp if
  // we wish
  newds.createTimeTags( dg, "NewData", 512, 256,
                        0, nanoSec(4) );

  // Create the new numeric data field 16 bits wide
  err = dataLE.createIntegralData( newds, "NEW_DATA", 16 );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Open the text data file for input
  FILE *datFile = fopen( dat, "r" );
  if( ! datFile )
  {
    io.printf( “Can’t open data file.” );
    return;
  }


  // step through original data, set time, read
  // in data from the text file in hex format,
  // then write the data using the replaceNext()
  // function

  while( ds.next( time ) )
  {
    newds.setPosition( time );
    fscanf( datFile, “%x”, &readValue );
    dataLE.replaceNext( readValue );
  }

  fclose( datFile );
  dg.setTimeCrossCorrelation();

}
```

File I/O: Writing to a File on the Disk

At times you may want to write to a file on the disk from within the Tool Development Kit program. You may accomplish this using standard C functions.

This example demonstrates writing to a file on the disk using the fprintf() function.

1 Open the /logic/demo/ToolDevKit/sample14.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 View the newly created file through the Tool Development Kit editor by opening the file "/logic/demo/ToolDevKit/data_out.txt" as a source file. The values written to this file correspond to the DATA values found whenever the STAT label value is 0xE.

In this example note that standard C formatting for the fprintf() function is used.

Files Used:

From the /logic/demo/ToolDevKit/ directory.

- sample14.___ (config file)
- sample.dat (Data file used by File In tool)
- sample14.c (Tool Development Kit program file)
- data_out.txt (Data file written to during file I/O operation)

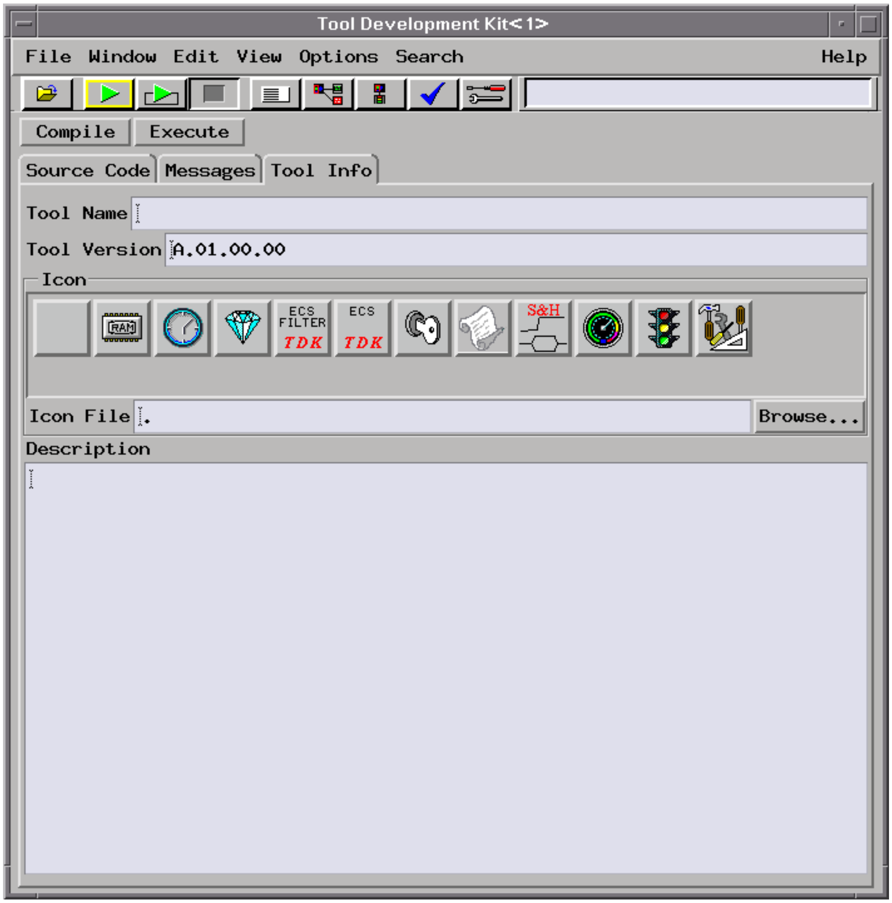
sample14.c

```
/*  File:  sample14.c
    Purpose:  Demonstrate file I/O by writing data to a
              file on the disk
*/


void execute(TDKDataGroup &dg, TDKBaseIO &io)
{
  // Define variable name for the text data file
  const char *dat =
     "/logic/demo/ToolDevKit/data_out.txt";

  // Open the text data file for output
  FILE *dataFile = fopen( dat, "w" );

  if( ! dataFile )
  {
    io.printf( "Can't open data file." );
    return;
  }

  // Declare other program variables
  TDKDataSet ds;
  TDKLabelEntry statLE;
  TDKLabelEntry dataLE;
  unsigned findValue = 0xE;  //hex value to search for
  unsigned searchValue;
  unsigned dataValue;

  // variable for keeping track of error codes
  int err;

  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  statLE.attach( ds, "STAT" );
  dataLE.attach( ds, "DATA" );
  // step through original DataSet, when the search
  // value is found in the STAT label, write out the Data
  // to the dataFile

  while( statLE.next(searchValue) && dataLE.next(dataValue) )
  {
    if( searchValue == findValue )
    {
      io.printf( “Found: %X”, dataValue );
      fprintf( dataFile, “ %X \n”, dataValue );
    }
  }

  fclose( dataFile );
}
```

Creating Installable Tools

Once your program has been tested and debugged using the Tool Development Kit, you can make an installable tool out of the program. This tool, which may be put on a floppy disk, or a local or networked directory, can be installed on any Agilent Technologies 16600 or 16700 series logic analyzer, allowing others to use the tool. The analyzer must be running version A.01.40.00 of the operating system or later.

NOTE: Installable tools created on a floppy disk MUST be installed from a floppy disk. Installable tools created on the file system (hard disk or mounted file system) MUST be installed from the file system.



Tool Name

Tool Version A.01.00.00

Icon File .

Description

Tool Development Kit Tool Info window

There are several steps that must be followed to create an installable tool. Assuming you have already done the hard part of coding the tool, the next step is to fill out the Tool Info window, found under the tool info tab of the Tool Development Kit. There are a few fields that must be filled out in this window.

Tool Name is the name the resulting tool will have. This will be the name of the tool as it will appear in the workspace.

Tool version is an identifier which tells you which version of the tool it is. This is a free-format version number.

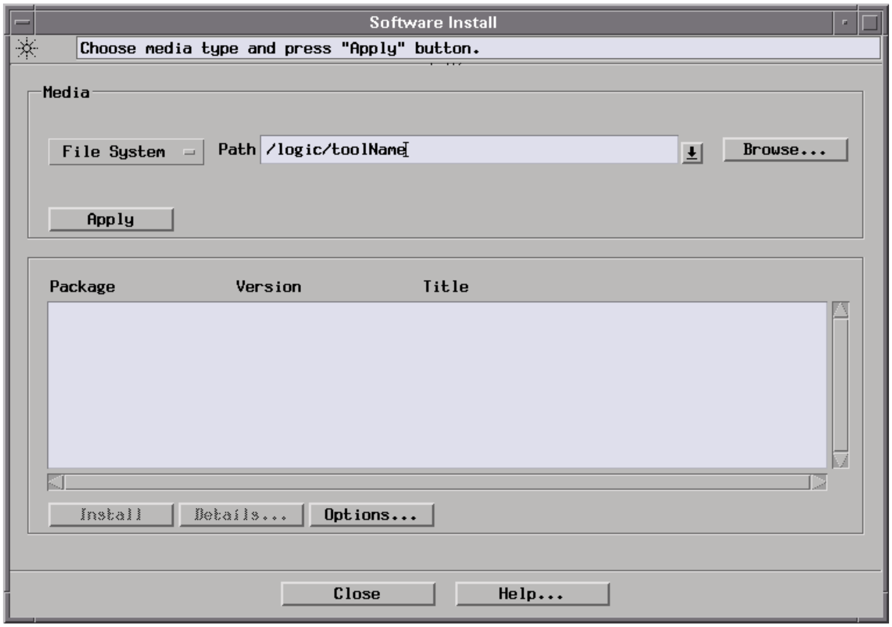
The icon may be chosen from the selection list in the Tool Info window. This is the icon that will appear on the workspace once the tool has been installed. If you wish you may create your own icon. The format must be 40x40 (pixel) pixmap. Pixmap is an Xwindows format. These can be created on x windows machines with xpaint, or converted from gifs on a PC. One freeware program which is available for the PC is called DISPLAY. This program can convert between many common graphics formats including pixmap (xpm). Simple modifications to pixmaps can be made by means of a text editor (including the Tool Development Kit text editor). The format uses an ascii representation of colors and pixels to make up the image. Once this has been created, the Browse button can be used to select the xpm file from the hard drive.

NOTE: The code must define the functions getDefaultArgs() and getLabelNames() because the resulting tool will rely upon these to create its interface. The tool will not work unless these are defined. See “Parameters” on page 187 for more information on using these functions.

Once all of this has been filled out, then the tool may be created. Use the File->Create Installable Tool... menu option to generate the tool. You may put the tool on the hard drive or floppy drive using the File Browser. A blank floppy should be used because the floppy disk will be re-formatted as part of the tool creation process. If more than one floppy is necessary, you will be prompted to insert additional floppies as necessary.

If the tool is to be installed directly from floppy, then it must be generated on the floppy at this step. If the tool is put into a file on the hard drive, then copied to a floppy, it will not be installable directly from the floppy disk. It is possible to copy the file back to a hard disk file and install from there, however.

The tool may be installed on a logic analyzer selecting “System Admin”, then “Software Install” tab, and then “Install...”. If the tool has been installed before, it is best to turn on the Option “reinstall even if same version exists”, to make sure that the tool will actually be re-installed. This is located under “Options...”. The tool may be installed from the floppy or the file system. Installing the tool will exit the current analyzer session.


Choose media type and press "Apply" button.

Path /logic/toolName

Software install window

Removing Tools

Tools may be removed from the analyzer after they have been installed. Using the Software install menu under the System Administration area, press the Remove button. A list of installed packages will appear. Select one of the packages and then push the remove button.

For best results in removing software, the session should be brought up clean. The tool to be removed should not in use, then after removal the session should be restarted.

Tool Versions and Config Files

Tools may be saved in config files. The name you give the tool is the identifier used by the config system to load the tool onto the workspace when the config is being loaded. Problems can happen if you make changes to the tool interface(i.e. its parameters) such that if a new version of the tool is installed and an old config is used to load the tool, there may be inconsistencies. For example, in the last version, the tool was programmed to look for a string in parameter 0, and in the new version it is looking for an int. Writing robust code can help to prevent fatal errors, but even the most robust code is incapable of predicting what future versions will bring.

If you create a tool that would break this compatibility, it is necessary to give it a new name. This will cause the config not to load the new tool. The user will be forced to create a new config saved with the new tool and will not run into any problems.

Creating an Installable Tool Tutorial

Any Tool Development Kit program can be converted into a tool once it has been tested. These tools can then be installed on any Agilent Technologies 16600 or 16700 series analyzer running version A.01.40.00 or later of the operating system. It is not necessary for the Tool Development Kit to be installed on any analyzer for an installable tool to operate properly.

Once created, a stand alone tool consists of the Parameters and the output window. The user will not necessarily know that the tool was created by Tool Development Kit.

This example demonstrates the procedure for creating a tool from the Mux.c program, and then how to use the tool.

NOTE: The Mux program is not discussed in detail here, but is included in the Extended Examples section of this document.

1 Open the /logic/demo/ToolDevKit/mux.___ config and the Tool Development Kit tool.

2 Make sure that the Tool Info window of the Tool Development Kit is complete for the mux tool. Select the “Tool Info” tab to enter the information. Be sure to give the tool a name and select an icon. See the section “Tool Info” on page 49.

3 Select File -> Create installable tool... from the Tool Development Kit window.

4 Select either the floppy disk or a file on the hard disk.

If it is desired to install the tool on multiple logic analyzers, it may be more convenient to create the installable tool on a floppy(s). The floppy(s) can then be used to install the newly created tool on to multiple logic analyzers.

If the tool will only be used by the logic analyzer that created the tool, it may be more convenient to create the installable tool from a file on the hard disk.

NOTE: Installable tools created on a floppy disk MUST be installed from a floppy disk. Installable tools created on the file system (hard disk or mounted file system) MUST be installed from the file system. Also note that all floppies used when creating an installable tool will be reformatted prior to copying the tool files on the floppy. You will loose all files and directories contained on the floppy during this process.

a If creating the installable tool on the hard disk drive, enter a filename.

If the installable tool was successfully created you will be given a message.

5 To install the newly created tool, open the System Admin dialog from the Logic Analysis System menu bar. Choose Software install, then press the Install button.

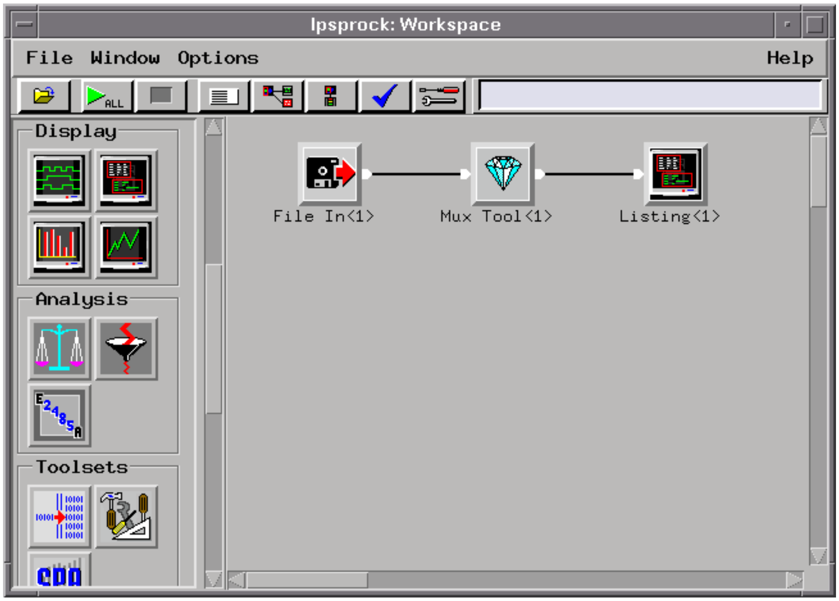
6 Using the file or floppy from step 3 and 4, select this as the install source in the Media window and then select Apply. The name of the tool should be shown in the Media window. The package name for the tool corresponds to the name of the original source code filename. In this case, mux will be the name of the package. The name entered into the Tool Info - Tool Name field will be the name of the tool (from step 2). Highlight the tool name and select Install. The tool will now be installed.

7 The session must be restarted before the tool will show up in the icon bar.

8 Drag the Mux tool onto the workspace replacing the Tool Development Kit tool.

9 Establish data flow through the Mux tool.

- **a** Drag from the output point of the File In tool to the input point of the Mux tool. Be sure to load the file "/logic/demo/ToolDevKit/muxdata_8.dat" in the File In tool.
- **b** Drag from the output point of the Mux tool to the input point of the Lister.



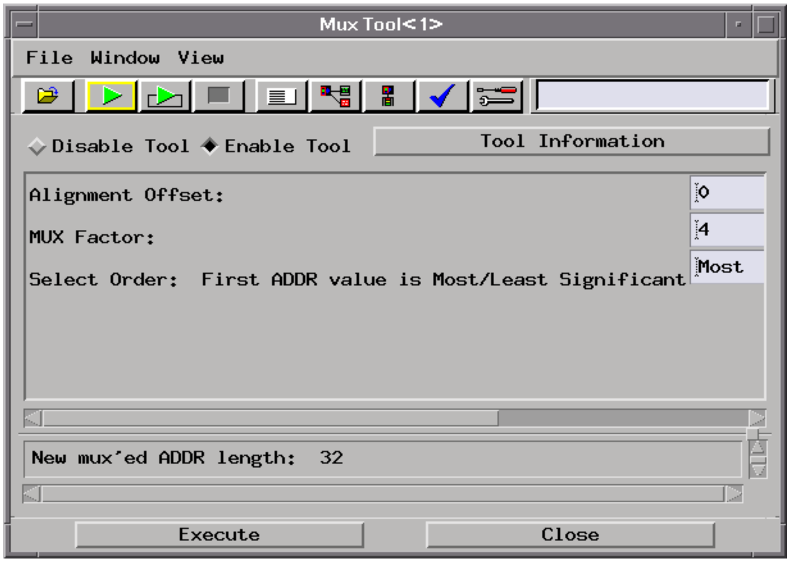
Main window showing the Mux tool

The workspace is now ready. To use the Mux tool:

10 Select the Mux tool icon and choose Display to open it.

11 Select the Run button.

12 Open the Lister at the output of the Mux tool to view the new DataSet.



◇ Disable Tool ◆ Enable Tool

Alignment Offset: 0

MUX Factor: 4

Select Order: First ADDR value is Most/Least Significant Most

New mux'ed ADDR length: 32

The opened Mux tool following a Run

You can see in this example that a tool can replace a Tool Development Kit program. The Mux tool uses a graphical user interface to get user input and to display messages. It also creates a new data set with modified data that can be viewed in the Lister.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- mux.___ (Configuration file)
- muxdata_8.dat (Data file used by File In tool)
- mux.c (Mux program file)

5

Tool Development Kit Programming Model

When you write code in the Tool Development Kit, be aware that in the model of Agilent Technologies 16600 and 16700 series analyzers, multiple copies of a tool may exist on the workspace. That is, you can drag out many listing tools, many filter tools, and many Tool Development Kit tools. This means that care must be taken that only local data is accessed by the program. Stay away from global variables and static local variables. The reason being that if more than one copy of a tool is put on the workspace, they all share the same global data. This is not the desired behavior in most cases.

Functions should also be given unique names. A potential problem is that two tools will define a function with the same name. This can cause unpredictable results. One way around this is to declare functions with the static specifier. This will ensure that the function has scope local to the file, and therefore only that tool will see the definition.

Sometimes it is useful for a Tool Development Kit program to accumulate data across multiple runs. This can best be achieved by appending data to a file.

The Tool Development Kit allows the user to write a program that is dynamically linked in to the same process as the system code. So, if your code has a bug that causes a crash, then the whole process will come down. Currently we are not able to support a stand-alone debugger with Tool Development Kit, so the best bet at this point is printf, and to prevent crashes from occurring in the first place. See “Debugging” on page 170.

Most of the examples for Tool Development Kit are written without using pointers. In most cases it is possible to use references, which are much safer and easier to program with. A reference parameter is used when the function is intended to change the value of the parameter

```
void incrementRef( int &x )
{
        x = x + 1;
}
```

The above function will add one to the value passed in. A reference parameter is essentially a pointer, but no special syntax is required to access the value that is pointed to. Here is the increment function implemented with standard pointers

```
void incrementPtr( int *x )
{
      *x = *x + 1;
}
```

Another difference is in how the two functions are called.

```
int x = 5;
incrementRef( x ); // now x is 6
incrementPtr( &x ); // now x is 7
```

Reference parameters can be used for efficiency as well. If it is not required that the value of the parameter be modified by the function, a const reference is used.

```
String concatStrings( const String &s1, const String
&s2 )
{
      return s1 + s2;
}
```

Demand Driven Tool

The Tool Development Kit tool is not a demand driven tool. Demand driven tools request a certain amount of data for any given analyzer run. The Tool Development Kit tool currently processes all of the data captured during any given run of the analyzer. There may be a significant amount of data that the Tool Development Kit tool must process. This will be dependent upon the size of data that can be captured by the acquisition module or card used by the logic analyzer. Be aware that processing this data may take a long time (minutes) through the Tool Development Kit tool. You may want to design your program so that you detect if the user has pressed the Cancel button. See “Interrupting the Run” on page 193 for information on how to do this.

Compiling Code

When you compile a program using the Tool Development Kit tool, it generates a number of “intermediate” files as part of the process. These files have the same name as the .c file that is being compiled, but have different extensions. The extensions are .i, .o, .sl, .ptrep. The .sl (shared library) is the executable file that is actually loaded into memory to run. All of the other files may be deleted at any time, but during development it is usually most convenient to leave these files in place to make the compilation process as efficient as possible.

Once development on a tool is complete, all the temporary files can be removed to save space on the disk.

NOTE: Compiling code on a mounted file system will slow the compile process. This is because all intermediate files will be created on that file system as well.

Compatibility

Since the Tool Development Kit produces tools that consist of compiled code, care must be taken to ensure that these tools will be compatible with future releases of the system code, compilers, OS, etc. The best way to do this is to follow the style of the examples presented in this manual and to not use any of the other C++ language features. These have a way of changing when compilers are revised.

Include Files

The Tool Development Kit only allows for single-file compilation, and does not allow for linking in additional libraries. You can however, include additional source files with the

```
#include “myfile.h”
```

construct. You do not need to worry about including the standard include files like stdlib.h, stdio.h, etc. They are automatically included.

Debugging

The Tool Development Kit does not support a debugger, but if you follow the recommended practices you will minimize the amount of crashes that you will encounter. Because the Tool Development Kit is based off the C language, it is possible to introduce serious bugs.

The primary method of debugging in the Tool Development Kit is to use io.printf and a "divide-and-conquer" technique to quickly find and deduce the region of the program that is causing the problems. Listed here are some common uses of io.printf that might be useful for debugging various variables used in a program. See the section on the I/O system for information on printing other items of interest.

- To print a *position* variable where position is of type long long (useful for functions that are iterating over data sets or label entries in order to know what state number or time sample is being referenced), use:

```
long long position;
io.printf( "The position is %lld",  position );
```

- To print information contained in some variable *myString* of type String use:

```
String myString;
io.printf( "The string is %s", (const char*)myString );
```

- To print a variable *myDouble* of type double use:

```
double  myDouble;
io.printf( "The value is %f", myDouble );
```

- To print a variable *myData* of type unsigned int use:

```
int myData;
io.printf( "My data value is %i", myData );
```

In general, integer values are formatted using the following conversion characters:

%d = signed integer
%i = signed integer
%o = unsigned integer using octal notation
%x = unsigned integer using lowercase hexadecimal notation
%X = unsigned integer using uppercase hexadecimal notation

- To print an array of characters use:

```
char buffer[100];
io.printf("The buffer contains %s", buffer);
```

- To print a character use:

```
char myCharacter;
io.printf("My character is %c", myCharacter);
```

If necessary, you can check the /logic/log/ directory to see a trace of what function was executing when the program crashed. The topmost function in this trace is the last one to be called. This is where the program crashed. However, that doesn't mean this is where the bug is. The bug may actually be in one of the calling functions.

A Note About the Functions

The functions below do not have any side effects. That is, the only variables that are modified by the call, are those that are directly involved in the function call itself, reference parameters, and the variable that is used to call the function (*ds* below). For example the call

```
ds.setPostion(pos);
```

does not effect the position of any of the labels in *ds*, or any other data set. Since *pos* is not passed by reference in this case, *ds* is the only variable which is affected. In general, variables are independent of one another; an operation on one has no effect on any other.

NOTE: Many of these functions return error codes. The value returned will be 0 for no error, and non zero for error. The code can be displayed as a string to the Runtime window using the functions in “Error Messages” on page -187.

Tool Development Kit Good Programming Practices and Helpful Hints

This section contains a list of good Tool Development Kit programming practices and in general helpful hints to use.

- Perform error checking on each function that provides it.

Many of the functions in the Tool Development Kit library return an error code, which is an integer representing the success of the call. See "Error Messages" on page -187 for more information.

- Only save configurations that are known to contain valid working Tool Development Kit programs.

Before saving a workspace that contains both a Tool Development Kit program and a data stream into the Tool Development Kit tool, check that the program compiles and runs without crashing or hanging. Whenever a configuration is loaded that contains data coming into a Tool Development Kit program, the "execute" function is immediately called. If the program contains an error that causes a crash or hang, the result will be another crash or hang without the opportunity to investigate or fix the problem.

- Use the View->datagroup... option found on the Tool Development Kit menu bar to access information about the incoming data group.

This dialog shows the number and name of each data set contained within the data group *dg* passed into the Tool Development Kit tool. In addition, it shows the number of samples contained in each data set. For each data set, the number and name of each label entry is shown. Expand the data set to see this information. For each label entry, the width in bits and the type of data is shown. All of this information can be retrieved by using the appropriate TDKDataGroup, TDKDataSet, or TDKLabelEntry function but you can also get it from this dialog.

NOTE: The name of the data set shown in the View->datagroup... dialog is not the entire name (origin name) of the data set. If you need the full name of the data set in order to attach the data set to a variable, use the function dg.getDataSetNames().

- When iterating over a data set or label entry, check that you don't step beyond the end of valid data.

The following data set and label entry iteration functions return 1 if data is valid else they return 0 for invalid data. Always check the value returned to ensure you are dealing with valid data.

```
int ds.next(long long &t)
int ds.next( )
int ds.prev(long long &t)
int ds.prev( )
int ds.peekNext(long long &t)
int ds.peekNext( )
int ds.peekPrev(long long &t)
int ds.peekPrev( )
int ds.replaceNext(long long &t)
int ds.replacePrev(long long &t)

int le.next(unsigned int &data)
int le.next(String &data)
int le.next(double &data)
int le.next(unsigned int &data, long long &pos)
int le.next(String &data, long long &pos)
int le.next(double &data, long long &pos)
int le.next( )

int le.prev(unsigned int &data)
int le.prev(String &data)
int le.prev(double &data)
int le.prev(unsigned int &data, long long &pos)
int le.prev(String &data, long long &pos)
int le.prev(double &data, long long &pos)
int le.prev( )

int le.peekNext(unsigned int &data)
int le.peekNext(String &data)
int le.peekNext(double &data)
int le.peekNext(unsigned int &data, long long &pos)
int le.peekNext(String &data, long long &pos)
int le.peekNext(double &data, long long &pos)

int le.peekPrev(unsigned int &data)
int le.peekPrev(String &data)
int le.peekPrev(double &data)
int le.peekPrev(unsigned int &data, long long &pos)
int le.peekPrev(String &data, long long &pos)
int le.peekPrev(double &data, long long &pos)
int le.replaceNext(unsigned int &data)
int le.replaceNext(String &data)
int le.replaceNext(double &data)

int le.replacePrev(unsigned int &data)
int le.replacePrev(String &data)
int le.replacePrev(double &data)
```

- Anytime new data sets have been created through the Tool Development Kit tool using any of the following functions:

  ds.createState
  ds.createTimeTags
  ds.createTimePeriodic

it is important that either dg.setTimeCrossCorrelation or dg.setStateCrossCorrelation be called from within the Tool Development Kit program. This informs downstream tools that either time or state correlated data is available.

- Use the following code to programmatically retrieve the list of data set names from a data group. This is helpful if you wish to attach a data set to a data group and there is more than one data set in the data group. This example also shows how to attach each data set in a group to a new data set variable. This example assumes there will not be more than 5 data sets.

```
int k;
int err;
int numDataSets;
StringList names;

dg.getDataSetNames( names );
numDataSets = dg.getNumberOfDataSets();
TDKDataSet ds[5];

for (k = 0; k < numDataSets; k++)
{
    io.print( names[k]);
    if ( k <= 4 ) // assume no more than 5 data sets
    {
        err = ds[k].attach(dg, names[k]);
        if (err)
        {
            io.printError( err );
            return;
        }
    }
}
```

If desired, you could retrieve the list of data set names similar in manner as the example above. Next use the interactive input to display the list of names with a number beside each and ask the user to input the number associated with the data set name they wish to use in the program. Then only attach to that data set. See the section “Interactive Input” on page 181.

Troubleshooting

This section contains a list of troubleshooting hints to use when encountering problems.

- Your newly created data set doesn't seem to line up correctly with the original data set when viewing them both in the Listing tool.

  Check that when the data set was created, the correlation time offset or correlation state offset retrieved from one of the input data sets to the program was passed into the create data set function. See the section on "TDKDataSet Creation Functions" on page 66 for more information.

- I created a new data set using the function createTimeTags and modified the time stamps, but not all of the values I wrote into label entries for the new data set show up correctly.

  Check that you have adhered to the rule that data sets must contain monotonically increasing time stamps. Ensure that you modified each samples time stamp in the new data set. See the TDKDataSet Creation Functions on page 66 for more information. Also ensure that the correct correlation time offset or correlation state offset was passed in.

- I am putting the correct value into the data set for a time value but when I use the "io.printf" to print it out for debug purposes, the value doesn't look right.

  Check that you have specified the correct formatting character when using the io.printf function. Time is stored in a "long long" type and requires the "%lld" formatting character. See the section on "Formatted output" on page 180.

- I created an installable tool and after I installed it, it core dumps.

  Did you remember that you must define the two functions "getDefaultArgs" and "getLabelNames"? The tool will not work unless these are defined.

- I get funny or changing results in the Listing display.

  If "createTimeTags()" was used, make sure the trigger offset value falls within the sample value.

6

Tool Development Kit System Utilities

This chapter discusses various Tool Development Kit system utility functions.

I/O System

The following functions defined for TDKBaseIO variables, which is passed into the tool through the execute routine, are used for interacting with the user and for creating user interfaces for the stand-alone tools. Note that it is not necessary to append a newline character in each print statement. The Tool Development Kit automatically does this.

Formatted output

io.printf

```
io.printf(const char *fmt, ...)
```

This is just like the standard printf function, except its output goes to the "Output" window. To print a long long use the following example.

```
long long number;
number = 0x0FFFFFFFFFFFFFFF;
io.printf( "Really big number = %lld", number );
```

CAUTION: Care must be used when using printf to print Strings. Strings must be cast to (const char *) in order for printf to function correctly. Here is an example.

```
String s;
s = "Hello World";
io.printf( "%s", (const char *) s );
```

It may be easier to call the following function for Strings.

io.print

```
void io.print( String msg )
```

This function simply sends the String msg to the Output window.

Printing from within functions other than "execute"

In order to print messages to the Output window from functions other than "execute", the *io* parameter passed into the "execute" function must also be passed into the specific function. The example below shows how to do this.

```
void myFunction( TDKBaseIO& io);

void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
   myFunction( io );
}

void myFunction( TDKBaseIO& io)
{
   io.printf("Inside of myFunction");
}
```

io.displayDialog

```
io.displayDialog( String msg )
```

This function simply displays the given string in a dialog. The dialog will only have the string and an 'OK' button. There is not a return value from the dialog.

Interactive Input

io.getUserInput

```
void io.getUserInput( const String& msg, String&
userReply, int& response )
void io.getUserInput( const String& msg, int& response )
```

These functions can be used to get input from the user of a tool interactively as the Tool Development Kit runs. In its first form, getUserInput() displays the prompt string *msg* in a dialog box and allows the user to enter a response, which is placed in *userReply*, a string reference parameter. Depending on whether the user pushes the

OK or Cancel button, the value of *response*, which is an int reference parameter will be 1 or 0 respectively. The value of *userReply* is used as the default for the input field of the dialog. If you wish for the input field to contain a predetermined value, then set *userReply* to the desired value.

In its second, simpler form, getUserInput() does not provide an input field, only Yes and No buttons. The prompt string is displayed as in the first form. The only way to get the user's response, depends on whether the Yes or No button is selected, which will cause *response* to get the value 1 or 0, respectively.

Next the tutorial will be presented showing how to use interactive input in a Tool Development Kit program.

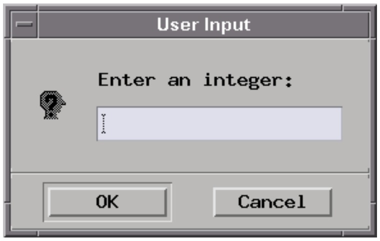
Using Interactive Input

There are times when you may need to get input from the user of your Tool Development Kit program. One way you can accomplish this is by displaying a dialog box to allow the user to enter a response.

This example demonstrates the use of the getUserInput() function: Displaying a dialog box with a prompt message and reading in user input.

1. Open the /logic/demo/ToolDevKit/sample12.___ config and the Tool Development Kit tool.
2. Select the Compile button, then Run.
   You will be prompted to enter an integer.
3. View the modified data in the Lister. Whenever the search value 0xF794 is found for the ADDR label, multiply the original DATA value by the value entered in the input dialog, then copy this into DATA2 label. This is seen at states 88 through 91.

4 You may view the original data by selecting the Lister icon at the output of the File In tool and choosing Display.


User Input

Enter an integer:

The user input dialog box

In this example the user is prompted for a response that is read in as a string, then converted to an integer. The second form of the getUserInput() function, not shown here, provides Yes and No selection buttons.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample12.___ (config file)
- sample.dat (Data file used by File In tool)
- sample12.c (Tool Development Kit program file)

sample12.c

```
/*  File:  sample12.c
    Purpose:  A TDK program to demonstrate interactive
              input -- the getUserInput() function.

    Create a new LabelEntry called DATA2.
    When searchValue it is found, multiply the original
    DATA value by searchValue, then copy this value
    into DATA2.

*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry addrLE;
  TDKLabelEntry dataLE;
  TDKLabelEntry data2LE;
  TDKDataSet ds;

  unsigned addrValue;
  unsigned dataValue;
  unsigned data2Value;
  unsigned searchValue = 0xf794;  //value to search for
  int multConst;   // integer used as a multiplier
  int resp;        //resp will hold the return value
                   // from the getUserInput() function
  String userResponse;
  String Prompt;

  // variable for keeping track of error codes
  int err;

  // Prompt user to input a decimal integer
  Prompt = “Enter an integer:  “;
  io.getUserInput(Prompt, userResponse, resp);

  // verify valid to user input
  if (!resp)
  {
    io.printf(“ Invalid user response. “);
    return;
  }

  // Assumes the user has entered an integer value in arg 0

  int num = 0;

  num = sscanf( userResponse, “%i”, &multConst );

  if( num != 1 )
  {
    io.printf( “Non-integer response.” );
    return;
  }

  // Attach to the incoming dataset
  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the “ADDR” label
  err = addrLE.attach( ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the DATA label
```

```
  err = dataLE.attach( ds, “DATA” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }


  // Create a new numeric data label, DATA2 which is
  // 16 bits wide, in the same DataSet
  err = data2LE.createIntegralData( ds, “DATA2”, 16 );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Loop through the ADDR and DATA values and multiply
  // the DATA value by a constant when the ADDR search
  // value is found

  while( addrLE.next( addrValue ) && dataLE.next( dataValue ) )
  {
    // Copy the dataValue
    data2Value = dataValue;

    // See if this is the desired ADDR value
    if( addrValue == searchValue )
    {
      data2Value *= multConst;
    }

    data2LE.replaceNext( data2Value );
  }

}
```

Error Messages

io.getErrorMessage

```
String io.getErrorMessage( int code )
```

Returns the string representation of an error code return value.

io.printError

```
void io.printError( int code )
```

Given an error code returned from one of the Tool Development Kit functions, this function will display the message corresponding to that error code in the Runtime window. You should not depend on a particular error code having the same meaning in future releases of the tool. Instead, use the above function to let the software tell you what the error is.

Parameters

Parameters are accessed through the following function.

io.getArg

```
String io.getArg(int n)
```

This function is used to retrieve the *nth* argument of the View Parameters... window. The argument list is zero (0) based; for example, io.getArg(0) gets the first argument defined. There is a limit of 50 parameters. It is up to the programmer to make sure the correct argument is being accessed. Additionally, the programmer can define how the parameters will be displayed on screen with the function getLabelNames() that can be defined in the program. This function is called by the system and tells it how to label the arguments. Here is an example.

```
StringList getLabelNames()
{
   StringList labels;
```

```
    labels.put("Clock label name");
    labels.put("Data label name");
    labels.put("Clock edge +/- = 1/0");
    labels.put("Setup range (ps) 0 to");
    labels.put("Hold range (ps) 0 to");
    labels.put("Setup violation (ps) >");
    labels.put("Hold violation (ps) >");
    labels.put("Stop on violation (off/on)");
    labels.put("Clock qualifier (off/on)");
    labels.put("Qualifier label name");
    labels.put("Qualifier value");
    return labels;
}
```

getDefaultArgs

Similarly, the function getDefaultArgs can be defined to setup defaults for each of the arguments. This way the user will be presented with a reasonable value that she may optionally change, if necessary.

```
StringList getDefaultArgs()
{
    // set default input parameters
    StringList defs;
    defs.put("CLOCK");
    defs.put("DATA");
    defs.put("1");
    defs.put("30000");
    defs.put("30000");
    defs.put("4000");
    defs.put("20000");
    defs.put("off");
    defs.put("off");
    defs.put("QUALIFIER");
    defs.put("1");
    return defs;
}
```

Parameters are always interpreted as Strings, so they must be converted to numeric types if that is how they are to be used. The standard library function sscanf() is useful for this purpose. Here is an example of inputting a long long value.

```
long long state_min;
int num =
sscanf( io.getArg( 1 ), "%lli", &state_min );
if( num != 1 )
{
     io.print( "Error converting state min" );
```

```
    return;
}
```

The above code will translate an integer in hex, octal, or decimal. It will return an error for non-matching input.

Next a tutorial will be presented showing how to use the parameters feature in a Tool Development Kit program.

Using Parameters

There are times when you may want to get input from the user of your Tool Development Kit program. User input (which is interpreted as string values) may be passed to the Tool Development Kit through the use of the View Parameters menu option.

This example demonstrates the use of parameters: creating label names for run-time arguments, assigning default run-time argument values, and reading input from the user.

1 Open the /logic/demo/ToolDevKit/sample11.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run in the Tool Development Kit tool.

3 View the modified data in the Lister.

- **a** Scroll to view the modified data. Whenever the search value 0xF794 is found for the ADDR label, multiply the original DATA value by the search value, then copy this into DATA2 label. This is seen at states 88 through 91.

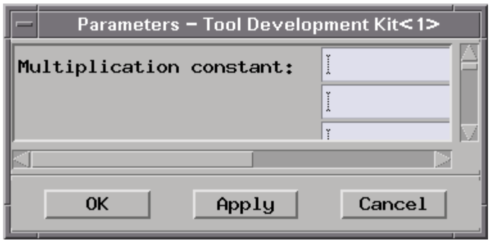
4 Use the run-time arguments.

- **a** Select View ➔View Parameters...
- **b** Enter a new integer value to be used as a multiplication constant.
- **c** Selec the Apply button.

d Select Run in the Tool Development Kit window.

5 Select the Lister window to see the newly modified data.

a The same states 88 through 91 will reflect new DATA2 values based on the multiplication constant entered through the View Parameters... window.



The View Parameters window

In this example, note that parameters are string values. Strings must be converted to integers using the sscanf() function if you need integer values or other numeric values.

Files Used: From the /logic/demo/ToolDevKit/ directory.

- sample11.___ (Config file)
- sample.dat (Data file used by File In tool)
- sample11.c (Tool Development Kit program file)

sample11.c

```
/*  File:  sample11.c
    Purpose:  A TDK program to demonstrate the use of
              runtime arguments

    Create a new LabelEntry called DATA2.
    When searchValue it is found, multiply the original
    DATA value by searchValue, then copy this value
    into DATA2.

*/


void execute(TDKDataGroup& dg, TDKBaseIO& io)
{

  TDKLabelEntry addrLE;
  TDKLabelEntry dataLE;
  TDKLabelEntry data2LE;
  TDKDataSet ds;

  unsigned addrValue;
  unsigned dataValue;
  unsigned data2Value;
  unsigned searchValue = 0xf794;
  int multConst;   // integer used as a multiplier

  // variable for keeping track of error codes
  int err;

  // Convert the parameter string to an integer.
  int num_read = sscanf( io.getArg(0), “%d”, &multConst );

  if( num_read == 0 )
  {
    io.print( “Error converting Parameter 1 to an integer:
Please re-enter” );
    return;
  }

  // Attach to the incoming dataset
  err = ds.attach( dg );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the “ADDR” label
  err = addrLE.attach( ds, “ADDR” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Attach to the DATA label
  err = dataLE.attach( ds, “DATA” );

  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Create a new numeric data label, DATA2 which is 16 bits
  // wide, in the same DataSet
  err = data2LE.createIntegralData( ds, “DATA2”, 16 );
  if( err )
  {
    // if we received an error, then print the message and exit
    io.printError( err );
    return;
  }

  // Loop through the ADDR and DATA values and multiply
  // the DATA value by a constant when the ADDR search
  // value is found

  while( addrLE.next( addrValue ) && dataLE.next( dataValue ) )
  {
    // Copy the dataValue
    data2Value = dataValue;

    // See if this is the desired ADDR value
    if (addrValue == searchValue)
    {
      data2Value *= multConst;
    }

    data2LE.replaceNext( data2Value );
  }
}


// Create label name for runtime argument
StringList getLabelNames()
{
  StringList labels;
  labels.put( “Multiplication constant:  ” );
  return labels;
}


// Assign default runtime argument
StringList getDefaultArgs()
{
  StringList defs;
  defs.put( “2” );
  return defs;
}
```

Interrupting the Run

If the program is complicated or it processes a lot of data, it may be useful for the user to be able to stop the execution through pressing the Cancel key. The function checkForUserAbort() is provided for this reason.

io.checkForUserAbort

```
int io.checkForUserAbort()
int io.checkForUserAbort( String msg )
```

Whenever it is called from the program, the Cancel key is checked. It is time consuming to do this, so care must be taken that the function is called only when necessary. It returns true if the user pressed Cancel. When given a String parameter, that string is displayed in the message area of the tool.

```
while( le.next( x ) )
{
   if( i % 100 == 0 )
   {
      if( io.checkForUserAbort()
      {
         return;
      }
   }
}
```

The above example checks the Cancel button once every 100 times through the loop. Checking every time is prohibitively expensive.

io.stop

```
void io.stop()
```

The Tool Development Kit can stop a repetitive run. This is achieved through the stop() function. This function will cause the analyzer to stop running repetitively. The tool will continue to execute code as normal, but will not execute again. This function can be useful for stopping when a certain condition has been detected. It has no effect if the analyzer in single run mode.

Time Convenience Functions

There are several built-in Tool Development utility functions to deal with converting time to various units. The logic analyzer is capable of showing time in picoseconds, nanoseconds, microseconds, milliseconds, and seconds. The following table shows the relationship between the various time units.

Time Units	Scientific Notation	Decimal Notation
1 second	1.0	1.0
1 millisecond	1.0 X 10^{-3} seconds	.001 seconds
1 microsecond	1.0 X 10^{-6} seconds	.000001 seconds
1 nanosecond	1.0 X 10^{-9} seconds	.000000001 seconds
1 picosecond	1.0 X 10^{-12} seconds	.000000000001 seconds

Time Units

Time unit functions all take a double parameter that represents a time value and converts the value to a long long type in picoseconds. Picoseconds are used by most library functions that deal with time. These functions are useful for conveniently dealing with other units of time.

picoSec

```
long long picoSec(double t)
```

This function takes the argument that represents a time in picoseconds and converts it to a long long type in picoseconds.

nanoSec

```
long long nanoSec(double t)
```

This function takes the argument that represents a time in nanoseconds and converts it to a long long type in picoseconds.

```
ds.setTimeBias();
/* ... other processing ... */
ds.setPosition(nanoSec(34.678));
```

microSec

```
long long microSec(double t)
```

This function takes the argument that represents a time in microseconds and converts it to a long long type in picoseconds.

milliSec

```
long long milliSec(double t)
```

This function takes the argument that represents a time in milliseconds and converts it to a long long type in picoseconds.

sec

```
long long sec(double t)
```

This function takes the argument that represents a time in seconds and converts it to a long long type in picoseconds.

timeToString

```
String timeToString(long long t)
```

This function takes the argument that represents a time in picoseconds and converts it to a string that is formatted in terms of scientific units similar to the way time is displayed in the lister tool.

```
ds.setTimeBias();
/* ... other processing ... */
print timeToString(ds.getPosition());
```

Utility Data Types

Two utility data types are included in the Tool Development Kit. The two types are Strings and Lists.

String Type

The String type is a more convenient type than const char *. It is unlimited in length, and for the most part, is compatible with const char *. It is necessary to cast to (const char *) in certain cases (e.g. the printf family of functions). Individual characters of a string may be accessed through the [] notation.

NOTE: A string variable is defined with the following syntax: String s;

Here are some of the useful functions defined for Strings.

s.chunk

```
String s.chunk( int offset, int length)
```

This function returns the substring of *s* starting *offset* chars from the beginning and *length* character long.

```
If s = "abcdef", then s.chunk( 2,3) is "cde".
```

s.length

```
int s.length()
```

Returns the length of *s*.

s.is_empty

```
int s.is_empty()
```

Returns true if and only if *s* is an empty string.

s.shrink

```
void s.shrink(int size)
```

Shrink the size of *s* to size. The rest of the string is gone.

int_to_str

```
String int_to_str( int x )
```

Converts x to a string. sscanf() can be used for more elaborate string formatting.

s1 + s2

```
s1 + s2
```

Concatenates the two strings.

List Type

The List type is used with StringList, and TDKLabelEntryList. In each of these cases the root type (String or TDKLabelEntry) is represented in a linked list structure. The root type can be manipulated with the following set of functions defined for Lists.

A List may be defined as follows:

```
StringList sList;
TDKLabelEntryList leList;
```

List.put

```
void sList.put( x )
void leList.put( y )
```

Adds the String x to the end of the list sList. Adds the TDKLabelEntry y to the end of the list leList.

List[n]

```
sList[n]
leList[n]
```

Returns the *nth* (0 based) element of the list. It works like an array.

List.length

```
int sList.length()
int leList.length()
```

Returns the length of the list.

Given the following variables and the previously defined lists sList and leList:

```
String myString;
TDKLabelEntry myLabel;
int length;

sList.put( myString );
leList.put( myLabel );
```

Adds the element myString to the end of the list sList and adds the element myLabel to the end of the list leList.

```
sList[3]
leList[3];
```

Returns the third String element in the list sList and returns the third TDKLabelEntry element in the list leList.

```
length = sList.length();
length = leList.length();
```

Returns the length of the list sList and leList.

NOTE: Make sure that l[n] is a valid element in the list. This implies you know the length of the list you are dealing with. If you do not, use the l.length function to find out the size of the list prior to retrieving an item on the list.

7

Extended Examples

Overview

The examples in this chapter are intended to demonstrate real-world functionality. These examples are more complex than those in the previous tutorials.

It is typical for a user to want to post-process data acquisitions to make the data more easily understood. For example, you may need to combine small pieces of data into one larger block or break a large block of data into smaller pieces. Or, you may need to interpret a particular protocol.

The Mux example demonstrates combining small pieces of data into a larger block of data, in this case 8-bit ADDR values into a 32-bit ADDR value. The Automotive demo breaks 16-bit data values into smaller meaningful pieces and interprets those values. The Automotive example also demonstrates viewing the time correlated data with listers and charts.

The Mux Program

There are situations where you may need to combine, or multiplex, several data values into one wider value.

This example uses an input data file containing an ADDR LabelEntry of a given length. The user may input a value, muxFactor, which is used to calculate a new multiplexed (mux'ed) ADDR length. The new ADDR length equals muxFactor times the original ADDR length, and the new ADDR value consists of muxFactor number of the original ADDR values combined. The code is heavily commented and contains explanations of the program's functionality and implementation.

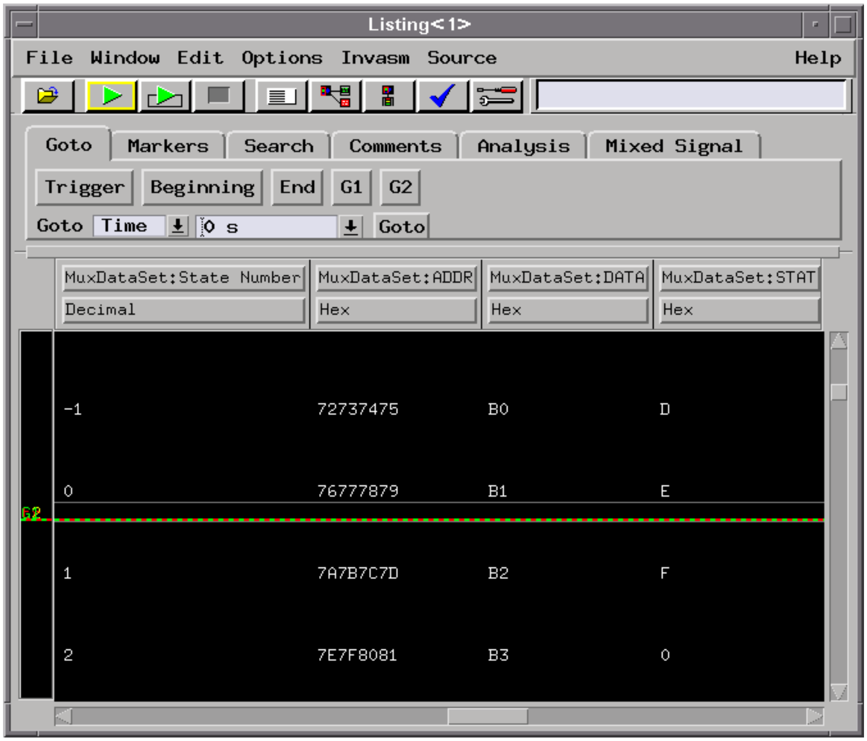
In the default configuration, the input data set has an 8-bit wide ADDR label and the Mux program creates a 32-bit mux'ed ADDR label. The user may enter different muxFactor values through the View Parameters menu option (so long as the maximum length of the new label does not exceed 32 bits). The user has the option of entering an alignment offset value which selects the position of the value to start with. The user may also choose the order in which ADDR values are combined: the first ADDR value encountered may be placed in the most or least significant position of the mux'ed ADDR value.

1 Open the /logic/demo/ToolDevKit/mux.__ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.

3 Open the Lister to view the original data set and the newly created Mux data set.

 a Use the horizontal scroll bar in the Lister window to view the new data set as shown in the following picture.



Listing<1>
File Window Edit Options Invasm Source Help
Goto Markers Search Comments Analysis Mixed Signal
Trigger Beginning End G1 G2
Goto Time 0 s Goto
G2

```
MuxDataSet:State Number  MuxDataSet:ADDR  MuxDataSet:DATA  MuxDataSet:STAT
Decimal                  Hex              Hex              Hex
-1                       72737475         B0               D
0                        76777879         B1               E
1                        7A7B7C7D         B2               F
2                        7E7F8081         B3               0
```

Lister resulting from running the Mux program

If you have created a tool from the Mux program (see the Creating a Tool tutorial example) and run the Mux program from the tool, you will see that the graphical user interface is used for user input and message display.

If you run the Mux program from the Tool Development Kit tool, you must use the menu option for user input. Output messages are displayed in the output window. Try entering different runtime arguments:

4 Select View→View Parameters...

5 Use the horizontal scroll bar in the View Parameters... dialog and scroll to enter one or more new arguments (MUX factor must not be greater than 4 so that the mux'ed ADDR length does not exceed 32 bits).

6 Select the Apply button.

7 Select Run in the Tool Development Kit window.

8 Select the Lister to view the modified DATA.

Files Used:

From the /logic/demo/ToolDevKit/ directory.

- mux.___ (Workspace file)
- muxdata_8.dat (8-bit input data file for the File In tool)
- mux.c (Tool Development Kit program file)

mux.c

```
/*  File:     mux.c

    Purpose:  This sample program uses an input data file containing
              ADDR, DATA, and STAT LabelEntries. The program
              multiplexes the ADDR LabelEntry by combining muxFactor
              number of ADDR values into one long long ADDR field.

              A new mux'ed DataSet is created containing LabelEntries
              of the same names as the original set. (Note: in order
              to use HP's inverse assemblers, ADDR, DATA, and STAT
              LabelEntries must exist in the same DataSet.)

              State 0 of the new multiplexed DataSet will occur at
              or before state 0 (time = 0) of the original DataSet.

              Using runtime arguments, the user enters choices for
              alignment offset, mux factor, and whether the most
              significant ADDR value encountered will be first in
              the new mux'ed field (most significant position) or
              last (least significant position).

              For example, a set of data contains an ADDR LabelEntry
              8 bits wide, and entries in this field need to be
              combined to form one 32 bit ADDR value. Here, the
              muxFactor is 4.  The alignment offset determines which
              ADDR value with respect to the first sample of the
              input data is the first value used in the new
              multiplexed ADDR value.  Samples are counted starting
              with 0.  If the alignment offset is 0, the first ADDR
              value (in the 0 position) is the starting value. If
              alignment offset is 2, the third value is the starting
              value ( position number 0, 1, 2).

              With an offset of 0 and the first ADDR value in the most
              significant position, we have for example:
```

```
              Original ADDR (8-bits)    Mux’ed ADDR (32-bits)

              FA
              FB
              FC
              FD                        FAFBFCFD

              FE
              FF
              00
              01                        FEFF0001


              The program then writes the original ADDR length, the
              muxFactor, and the mux’ed ADDR length to the output
              message window.

*/

void execute(TDKDataGroup& dg, TDKBaseIO& io)
{
  // Create variables for the original and mux DataSets
  TDKDataSet ds;
  TDKDataSet muxDS;

  // Create variables for each LabelEntry in both the input
  // and mux DataSets
  TDKLabelEntry addrLE;
  TDKLabelEntry dataLE;
  TDKLabelEntry statLE;

  TDKLabelEntry muxAddrLE;
  TDKLabelEntry muxDataLE;
  TDKLabelEntry muxStatLE;


  // Other program variables
  unsigned int addrValue;         // original ADDR field
  unsigned int muxAddrValue;      // mux’ed ADDR field
  unsigned int dataValue;         // original DATA field
  unsigned int statValue;         // original STAT field
  int i;
  int startRow;          // alignment offset from beginning of samples
  int muxFactor;         // how many original fields to combine
  int firstHi;           // first ADDR value takes the high position?
  int addrLen;           // original ADDR length, in bits
  int muxAddrLen;        // new mux’ed ADDR length, in bits
  int dataLen;           // length of DATA and field, in bits
  int statLen;           // length of STAT field, in bits
  int temp;
  int count;
  int modFactor;                 // original number of samples modulus
                                 // muxFactor
  int triggerRow;                // trigger row of mux DataSet
  unsigned int numSamples;       // number of samples in mux DataSet
  unsigned int origNumSamples;   // number of samples in original
                                 // DataSet
  long long time;                // time stamp for mux DataSet
  long long correlationTime;     // correlation time for DataSets


  // Assign runtime argument values, if the user has not
  // input runtime values, default values will be used.
  // Runtime arguments are strings, and must be converted if
  // numeric values are required

  int num = 0;

  num = sscanf( io.getArg( 0 ), “%i”, &startRow );

  if( num != 1 )
  {
    io.print( “Unable to convert start row parameter” );
    return;
  }

  num = sscanf( io.getArg( 1 ), “%i”, &muxFactor );

  if( num != 1 )
  {
    io.print( “Unable to convert mux factor parameter” );
    return;
  }

  // The user may enter “Most” or “Least” as runtime arguments
  // to indicate the order to combine ADDR values.
  // “Most” indicates that the first original ADDR value encountered
  // will be in the most significant position of the new mux’ed ADDR.
  // “Least” indicates that the first original ADDR value encountered
  //  will be in the least significant position of the new mux’ed ADDR.
  firstHi = (io.getArg(2) == “Most” );   // 1 = Most, 0 = Least

  int err = 0;

  // Attach to the incoming dataset
  err = ds.attach(dg);
  if( err )
  {
    io.printError( err );
    return;
  }

  correlationTime = ds.getCorrelationTime();

  // Time rather than state will be used to correlate DataSets
  ds.setTimeBias();

  // Attach to the ADDR label
  err = addrLE.attach(ds, “ADDR” );
  if( err )
  {
    io.printError( err );
    return;
  }

  // Attach to DATA label
  err = dataLE.attach(ds, “DATA” );
  if( err )
  {
    io.printError( err );
    return;
  }

  // Attach to STAT label
  err = statLE.attach(ds, “STAT” );
  if( err )
```

```
  {
    io.printError( err );
    return;
  }

  // Get information about the input DataSet
  addrLen = addrLE.getWidth();
  dataLen = dataLE.getWidth();
  statLen = statLE.getWidth();
  origNumSamples = ds.getNumberOfSamples();

  if( muxFactor <= 0 || muxFactor > 32 )
  {
    io.printf(
      “The given mux factor %i is out of range. Using mux factor = 4”,
        muxFactor );

    muxFactor = 4;
  }

  // Calculate values to be used with the mux DataSet
  muxAddrLen = addrLen * muxFactor;

  if( muxAddrLen > 32 )
  {
    io.printf(
       “The ADDR label width times the mux factor is greater than” );
    io.printf( “32.  Exiting. “ );

    return;
  }

  modFactor =  origNumSamples % muxFactor;

  numSamples = (origNumSamples - modFactor) / muxFactor;
  // Do not create too many samples in the new DataSet
  // since all states must be assigned values
  while (  ((numSamples * muxFactor) +  startRow ) > origNumSamples )
  {
     numSamples -= 1;
  }



/*
   Calculate trigger row for the mux DataSet:
   In order to provide for use of data filters, the input
   DataSet trigger point is found by stepping through
   the states keeping count, and when state 0 (time = 0)
   is found, that count is the trigger row.
*/

  // read the beginning time without incrementing the DataSet iterator
  ds.peekNext(time);

  // initialize triggerRow count
  triggerRow = -1;

  // Account for alignment offset
  for (i=0; i < startRow; i++)
  {
    ds.next(time);
  }
  // Step through the original DataSet in groups of size muxFactor
  // until time = 0 is found
  while (time <= nanoSec(0) )
  {
    for (int j = 0; j < muxFactor ; j++)
    {
      ds.next(time);
    }

    // If state 0 is passed in this group, don’t increment trigger row
    if(time <= nanoSec(0))
    {
      triggerRow++;
    }
  }


  // Reset the DataSet iterator to the beginning
  ds.reset();

/*
   Create the new mux’ed dataset.  The time value used for the
   samplePeriod parameter is a dummy value. The correct time tag
   is written in the while loop that assigns values to the new
   DataSet’s fields.
*/
  err = muxDS.createTimeTags(dg, “MuxDataSet”, numSamples, triggerRow,
                             correlationTime, nanoSec(4.0) );

  if( err )
  {
    io.printError( err );
    return;
  }

  // Create a new field label, ADDR, in the mux’ed DataSet with a width
  // equal to muxFactor times the original ADDR field width
  err = muxAddrLE.createIntegralData( muxDS, “ADDR”, muxAddrLen );
  if( err )
  {
    io.printError( err );
    return;
  }

  // Create a new field label, DATA, in the mux DataSet which is
  // dataLen bits wide DATA values from the original DataSet will
  // not be modified
  err = muxDataLE.createIntegralData( muxDS, “DATA”, dataLen );
  if( err )
  {
    io.printError( err );
    return;
  }

  // Create a new field label, STAT, in the mux DataSet which is
  // statLen bits wide STAT values from the original DataSet will
  // not be modified
  err = muxStatLE.createIntegralData( muxDS, “STAT”, statLen );
  if( err )
  {
    io.printError( err );
    return;
  }

  // skip past startRow number of states in the original
  // DataSet to get to the first state used in the
  // multiplexing operation
  for( i = 0; i < startRow; i++)
  {
    ds.next(time);
    addrLE.next();
    dataLE.next();
    statLE.next();
  }


  count = 0;
  temp = 0;
  muxAddrValue = 0;


/*
   Loop through the the original DataSet calculating values for the
   mux’ed set Assign ADDR, DATA and STAT values to the mux DataSet
*/

  while ( ds.next(time) && addrLE.next(addrValue) &&
          dataLE.next(dataValue) && statLE.next(statValue) )
  {
    // Build the new muxAddrValue over muxFactor number of iterations
    if (firstHi)    // first value of original ADDR values is high
    {
      temp = addrValue<<( (muxFactor-1-count) * addrLen);
      muxAddrValue =  temp | muxAddrValue;
    }
    else              // last value of original ADDR values is high
    {
      temp = addrValue<<( count * addrLen);
      muxAddrValue = temp | muxAddrValue;
    }

    // every muxFactor number of loops, write the new values for the
    // mux DataSet
    if( count == (muxFactor -1) )
    {
      // Write out the correct time stamp first
      muxDS.replaceNext( time );
      muxAddrLE.replaceNext( muxAddrValue );
      // clear muxAddrValue in preparation for next build
      muxAddrValue = 0;
      muxDataLE.replaceNext( dataValue );
      muxStatLE.replaceNext( statValue );
    }
    count = (count + 1) % muxFactor;
  }
// end of the while loop


  // Print out the new mux’ed ADDR length information
  // to the output window
  io.printf(“Original ADDR length:  %d”, addrLen);
  io.printf(“MuxFactor used:  %d”, muxFactor);
  io.printf(“New mux’ed ADDR length:  %d”, muxAddrLen);

  // tag the dataGroup so that downstream tools
  // know that the data is time correlated
  dg.setTimeCrossCorrelation();

}


////////////////  DO NOT REMOVE THE FOLLOWING  ////////////////
// The following functions may be modified but not removed,  //
// even if you are not getting parameters from the user.     //
// If these functions are removed, you will NOT be able to   //
// create installable tools.                                 //
///////////////////////////////////////////////////////////////

// Assign label names for runtime arguments in the
// order that they will appear
StringList getLabelNames()
{
  StringList labels;

  labels.put("Alignment Offset: ");
  labels.put("MUX Factor: ");
  labels.put(
         "Select Order:  First ADDR value is Most/Least Significant");

  return labels;
}


// Assign default values to runtime arguments
StringList getDefaultArgs()
{
  StringList defs;

  defs.put("0");      // alignment offset = 0
  defs.put("4");      // muxFactor = 4
  defs.put("Most");   // first ADDR value is most significant

  return defs;
}
```

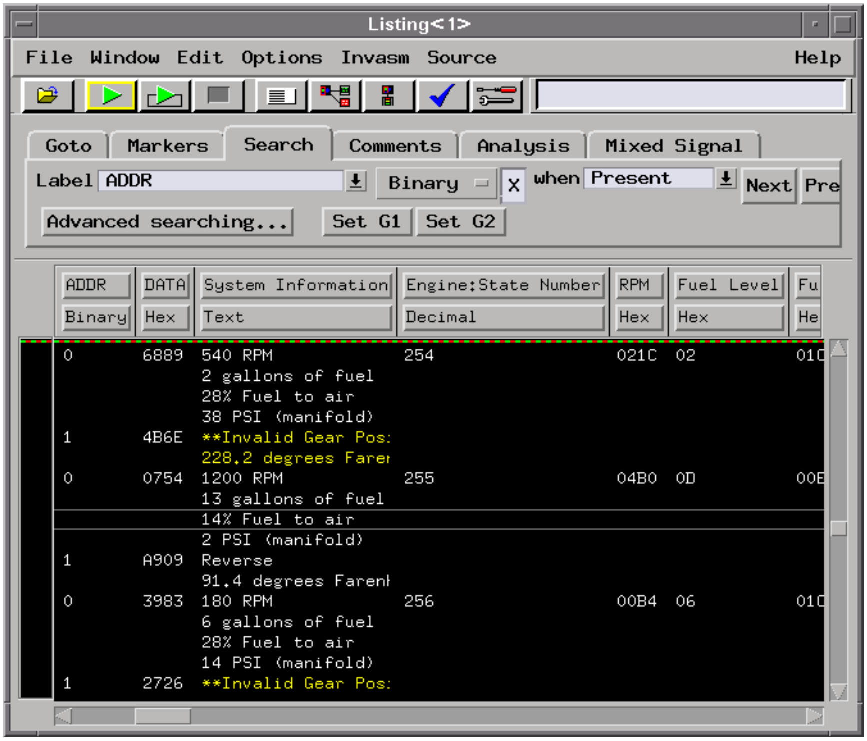
The Automotive Program

There are situations where you may wish to break apart a data value into smaller pieces and interpret those values.

The Automotive sample demonstrates breaking 16-bit data values into multiple data components, then interpreting and displaying those values. The input data set contains a 16-bit DATA label, which depending on a flag, may represent engine or transmission information. The program interprets the data, creates new data sets for that data, displays a text interpretation of the data, and highlights certain conditions in the data with color. The code is commented and contains detailed explanations of the program's functionality and implementation.

1 Open the /logic/demo/ToolDevKit/auto.___ config and the Tool Development Kit tool.

2 Select the Compile button, then Run.


Listing<1>
File Window Edit Options Invasm Source Help
Goto Markers Search Comments Analysis Mixed Signal
Label ADDR
Binary
X
when Present
Next Pre
Advanced searching...
Set G1 Set G2

```
ADDR    DATA  System Information   Engine:State Number  RPM   Fuel Level  Fu
Binary  Hex   Text                 Decimal              Hex   Hex         He
0       6889  540 RPM              254                  021C  02          010
              2 gallons of fuel
              28% Fuel to air
              38 PSI (manifold)
1       4B6E  **Invalid Gear Pos:
              228.2 degrees Farer
0       0754  1200 RPM             255                  04B0  0D          00E
              13 gallons of fuel
              14% Fuel to air
              2 PSI (manifold)
1       A909  Reverse
              91.4 degrees Farenh
0       3983  180 RPM              256                  00B4  06          010
              6 gallons of fuel
              28% Fuel to air
              14 PSI (manifold)
1       2726  **Invalid Gear Pos:
```

Displays from the Automotive demo

Files Used: From the /logic/demo/ToolDevKit/ directory.

- auto.___ (Workspace file)
- auto.dat (Data file for File In tool)
- auto.c (Tool Development Kit program file)

auto.c

```
/*  File:     auto.c

    Purpose:  The Auto sample demonstrates breaking 16 bit data values
             into multiple data components, interpreting and displaying
              those values, and coloring text associated with error
              conditions in order to make those states more easily
              visible. Resulting data is then displayed using charts
              which are time correlated with each other and with the
              listings.


              The input file, auto.dat, contains a single bit ADDR and
              16 bit DATA label with samples every 10mS.  This data is
              not actual automotive bus data but has been created just
              for demonstration purposes.

             When ADDR = 0, DATA represents engine related information:
                 the first 6 bits represent rpm’s (rpm);
                 the next 4 bits represent fuel level (fuelLevel);
                the next 3 bits represent fuel to air ratio (fuelAir);
               the last 3 bits represent manifold pressure (manifold).

              When ADDR = 1, DATA represents transmission information:
                 the first 3 bits represent gear position (gear);
                 the next 8 bits represent temperature (temp);
                 the remaining bits are unused.

              For example, the second sample contains an ADDR value = 0
              Which indicates engine data and DATA = 4691 which viewed
             in binary format is 0100 0110 1001 0001 .  Broken into its
              data components:


                 0 1 0       0 0 1     1 0 1 0    0 1 0 0 0 1
                 \   /       \   /     \     /    \         /
                 manifold   fuelAir   fuelLevel       rpm
                    2          1          10           17    (decimal)

              These raw data values are then used to calculate final
              values (data = the raw data value):

                manifold =  ( (data * 28.5) + 6.5 ) / 2.4 = 26
                  where the raw data value = DATA right shifted 13
                  places;

                fuelAir = data * 14.286 = 14
                  where the raw data value = DATA right shifted 10
                  places and masked ( bitwise AND’ed ) with 03 hex;

                fuelLevel = data = 10
                  data = DATA right shifted 6 places and masked with
                  0f hex;

                rpm = data * 60 = 1020
                 where the raw data value = DATA masked with 03f hex.



*/


// enumerated types to make engine vs transmission data
// and LabelEntry array access more clear
enum{engine, transmission};
enum{Address, Dat, SystemInfo} ;
// the following type names are abbreviated since
// the words rpm, manifold, fuelLevel and fuelAir
// are used as variable names
enum{RPMs, FLevel, FAir, Mani};
enum{GearPosition, Temperature};

enum { white, white2, scarlet, pumpkin, yellow, lime, turquoise,
lavender };
```

```
void execute(TDKDataGroup &dg, TDKBaseIO &io)
{
  // define LabelEntries and DataSets

  TDKLabelEntry le[3];
  // 0 = ADDR, 1 = DATA, 2 = System Information

  TDKLabelEntry engineLE[4];
  // 0 = RPM, 1 = Fuel Level, 2 = Fuel/Air, 3 = Manifold

  TDKLabelEntry transLE[2];
  // 0 = Gear, 1 = Temp

  TDKDataSet    ds;
  TDKDataSet    engineDS;
  TDKDataSet    transDS;


  // incoming ADDR value
  unsigned addr;

  // incoming DATA value
  unsigned data;

  // data components
  unsigned rpm;
  unsigned manifold;
  unsigned fuelLevel;
  unsigned fuelAir;
  unsigned gear;
  double temp;

  // define other program variables
  unsigned numSamples;      // number of samples in each new DataSet
  char message[100];
  String gearMessage;
  long long time;
  long long correlationTime;      // correlation time for DataSets
  int celsius;

  int err;

  String invalidStr;
  invalidStr = "**Invalid Gear Position**";

  // read runtime argument
  celsius = io.getArg(0) == "c" || io.getArg(0) == "C";

  // Attach to the incoming dataset
  err = ds.attach(dg);

  if( err )
  {
    io.printError( err );
    return;
  }

  correlationTime = ds.getCorrelationTime();

  // DataSets will be time correlated
  err = ds.setTimeBias();

  if( err )
  {
    io.printError( err );
    return;
  }

  // Attach to the address label
  err = le[Address].attach(ds, “ADDR” );

  if( err )
  {
    io.printError( err );
    return;
  }

  // Attach to the data label
  err = le[Dat].attach(ds, “DATA” );

  if( err )
  {
    io.printError( err );
    return;
  }

  // Create a new System Information label
  err = le[SystemInfo].createTextData(ds, “System Information”, 15 );

  if( err )
  {
    io.printError( err );
    return;
  }


/*   Create two new DataSets, one for ADDR = 0 :  Engine Statistics
 *   the other for ADDR = 1 :  Transmission Information
 *
 *   Each will have half the number of states as the input DataSet
 *   and will be time stamped for accurate time correlation.
 *   Creating the new DataSets in this way will allow us to use
 *   charts to display data for each category with proper time
.*   correlation.
 *
 *   The samplePeriod value used in both createTimeTags functions is
 *   a dummy value which will be replaced in the while loop
*/

  numSamples = ds.getNumberOfSamples() / 2;

  engineDS.createTimeTags(dg, “Engine”, numSamples, 0,
                          correlationTime, nanoSec(200.0) );
  transDS.createTimeTags(dg, “Trans”, numSamples, 0,
                         correlationTime, nanoSec(200.0) );


  // Create new LabelEntries
  engineLE[RPMs].createIntegralData(engineDS, “RPM”, 14);
  engineLE[FLevel].createIntegralData(engineDS, “Fuel Level”, 6);
  engineLE[FAir].createIntegralData(engineDS, “Fuel/Air”, 10);
  engineLE[Mani].createIntegralData(engineDS, “Manifold”, 10);

  transLE[GearPosition].createIntegralData(transDS, “Gear”, 3);
  transLE[Temperature].createIntegralData(transDS, “Temp”, 8);
```

```
/*   The main while loop:
 *
 *   This loop reads the next sample’s time, ADDR, and DATA.
 *   If ADDR = 0, then the DATA represents engine information.
 *   If ADDR = 1, then the DATA represents transmission information.
 *   Data is interpreted, written to the new DataSets, and the
 *   text interpretation message is formed.
*/

  while ( ds.next( time ) && le[Address].next( addr ) &&
                             le[Dat].next( data ) )
  {
    if ( addr == engine )          // Engine information
    {
      engineDS.replaceNext(time);

      // The first 6 bits indicate engine rpm’s
      rpm = ( data & 0x3f ) * 60;
      engineLE[RPMs].replaceNext(rpm);


      // The next 4 bits indicate fuel level
      fuelLevel = ( data >> 6 ) & 0xf;
      engineLE[FLevel].replaceNext(fuelLevel);

      // The next 3 bits indicate air ratio
      fuelAir = ( ( data >> 10 ) & 0x3 ) * 14.286;
      engineLE[FAir].replaceNext(fuelAir);

      // The next 3 bit indicate fuel to manifold pressure
      manifold = ( ( data >> 13 )  * 28.5 + 6.5 ) / 2.4;
      engineLE[Mani].replaceNext(manifold);

      // Format the results
      sprintf ( message, “%d RPM\n%d gallons of fuel\n%d%% “
                         “Fuel to air\n%d PSI (manifold)”,
                         rpm, fuelLevel, fuelAir, manifold );

    }

    if ( addr == transmission )          // Transmission information
    {
        transDS.replaceNext(time);

        // The first 3 bits indicate transmission position
        gear = ( data & 0x7 );
        transLE[GearPosition].replaceNext(gear);

          switch ( gear )
          {
            case 0:
              gearMessage = “Park”;
              break;
            case 1:
              gearMessage = “Reverse”;
              break;
            case 2:
              gearMessage = “Neutral”;
              break;
            case 3:
              gearMessage = “Overdrive”;      // or 3rd gear
              break;
            case 4:
              gearMessage = “2nd Gear”;
              break;
            case 5:
              gearMessage = “1st Gear”;
              break;
            default:            // invalid gear position
                                // set message color to yellow
              gearMessage = invalidStr;
              le[SystemInfo].setColor(le[SystemInfo].getPosition(),
                                      yellow);
              break;
        };

        // The next 8 bits indicate transmission temperature
        if(celsius)
        {
          temp = ( ( data >> 3 ) & 0xff );
          sprintf ( message, “%s\n%3.1f degrees Celsius”,
                    (const char *) gearMessage, temp );
        }
        else          // temperature is in Fahrenheit
        {
          temp = ( ( data >> 3 ) & 0xff ) * ( 9.0 / 5.0 ) + 32.0;
          sprintf ( message, “%s\n%3.1f degrees Farenheit”,
                    (const char *) gearMessage, temp );
        }

        transLE[Temperature].replaceNext(temp);

      }

    // prepare message
    String s;
    s = message;
    le[SystemInfo].replaceNext( s );
  }

  // this command must be used so that downstream
  // tools know that the data is time correlated
  dg.setTimeCrossCorrelation();

}


////////////////  DO NOT REMOVE THE FOLLOWING  ////////////////
// The following functions may be modified but not removed,  //
// even if you are not getting parameters from the user.     //
// If these functions are removed, you will NOT be able to   //
// create installable tools.                                 //
///////////////////////////////////////////////////////////////

// define label names for runtime arguments
StringList getLabelNames()
{
  StringList l;

  l.put(“ Temperature Units: F or C”);

  return l;

}
```

```
// set default values for runtime arguments
StringList getDefaultArgs()
{
  StringList l;

  l.put(“F”);

  return l;

}
```

8

Tool Development Kit Reference

This chapter discusses Tool Development Kit functions in alphabetical order.

I/O System Functions

getDefaultArgs

```
StringList getDefaultArgs()
```

This function is called by the system to get information from the program. The information is used to display default values in the View Parameters... window. Programs using the View Parameters... window to retrieve information at runtime must define and implement this function within the program. Use io.getArg(int n) to retrieve the *nth* parameter entered into the View Parameters... window. See “Parameters” on page -187 for more information on using this function.

getLabelNames

```
StringList getLabelNames()
```

This function is called by the system to get the names of the labels to be placed next to each input field in the View Parameters... window. A maximum of 50 label names may be defined. Programs using the View Parameters... window to retrieve information at runtime must define and implement this function within the program in order to put a label next to each input field. See “Parameters” on page -187 for more information on using this function.

io.checkForUserAbort

```
int io.checkForUserAbort()
```

This functions checks if the user has press the Cancel key. Returns true if they have, otherwise returns false.

io.checkForUserAbort

```
int io.checkForUserAbort( String msg )
```

This functions checks if the user has press the Cancel key. The String *msg* is displayed in the message area of the tool. Returns true if they have, otherwise returns false.

io.displayDialog

```
io.displayDialog( String msg )
```

This function simply displays the given string in a dialog. The dialog will only have the string and an 'OK' button. There is not a return value from the dialog.

io.getArg

```
String io.getArg(int n)
```

This function is used to retrieve the *nth* argument of the View Parameters... window. It is up to the programmer to make sure the correct argument is being accessed. Programs using getArg(int n) must have defined and implemented the functions getDefaultArgs and getLabelNames.

io.getErrorMessage

```
String io.getErrorMessage( int code )
```

Returns the string representation of an error *code* return value.

io.getUserInput

```
void io.getUserInput( const String& msg, int& response )
```

Used to get input from the user of a tool interactively as the Tool Development Kit runs. Displays the prompt string *msg* in a dialog box. Only allows the user to select the Yes or No buttons. The user's response depends on whether the Yes or No button is selected, which will cause *response* to get the value 1 or 0, respectively.

io.getUserInput

```
void io.getUserInput( const String& msg, String& input,
int& response )
```

Used to get input from the user of a tool interactively as the Tool Development Kit runs. Displays the prompt string *msg* in a dialog box and allows the user to enter a response, which is placed in *input*, a string reference parameter. Depending on whether the user pushes the OK or Cancel button, the value of *response*, which is an int reference parameter, will be 1 or 0 respectively. The value of input is used as the default for the *input* field of the dialog.

io.print

```
void io.print( String msg )
```

This function simply sends the String *msg* to the Output window.

io.printError

```
void io.printError( int code )
```

Given an error *code* returned from one of the Tool Development Kit functions, this function will display the message corresponding to that error code in the Runtime window.

io.printf

```
io.printf(const char *fmt, ...)
```

This is just like the standard printf function, except its output goes to the "Output" window.

io.stop

```
void io.stop()
```

This function stops the analyzer during a repetitive run.

List Functions

myList.length

```
int myList.length()
```

Returns the length of the list. myList is of type StringList or TDKLabelEntryList.

myList[n]

```
myList[n]
```

Returns the *nth* (0 based) element of the list. It works like an array. myList is of type StringList or TDKLabelEntryList. *n* is of type String or TDKLabelEntry (is the same type as the list.)

myList.put

```
void myList.put( x )
```

Adds the element *x* to the end of the list *myList*. *myList* is of type StringList or TDKLabelEntryList. *x* is of type String or TDKLabelEntry (must be same root type as the list.)

String Functions

int_to_str

```
String int_to_str( int x )
```

Converts x to a string. sscanf() can be used for more elaborate string formatting.

s.chunk

```
String s.chunk( int offset, int length)
```

This function returns the substring of *s* starting *offset* chars from the beginning and *length* character long.

s.is_empty

```
int s.is_empty()
```

Returns true if and only if *s* is an empty string.

s.length

```
int s.length()
```

Returns the length of *s*.

s.shrink

```
void s.shrink(int size)
```

Shrink the size of *s* to size. The rest of the string is gone.

s1 + s2

```
s1 + s2
```

Concatenates the two strings.

TDKCorrelator Functions

c.firstPosition

```
long long c.firstPosition()
```

This function will return the Time or State value of the first position for this correlator depending on the current bias. If the bias setting for *c* is State, then the value found under the State column at the position of the pointer *c* will be returned. If the bias setting for *c* is Time, then the value found under the Time column at the position of the pointer *c* will be returned. Time values are in picoseconds.

c.getPosition

```
long long c.getPosition()
```

This function will return the Time or State value of the current position for the correlator depending on the current bias. If the bias setting for *c* is State, then the value found under the State column at the position of the pointer *c* will be returned. If the bias setting for *c* is Time, then the value found under the Time column at the position of the pointer *c* will be returned. Time values are in picoseconds.

c.initialize

```
int c.initialize( TDKDataSet referenceDataSet)
int c.initialize( TDKDataSet referenceDataSet,
TDKLabelEntryList leList )
```

The initialize functions prepare the data sets to be correlated. In its second form, an array of label entries is given. *referencedataSet* is the data set which is used for position information in the correlation process. *leList* is a list of label entries to be correlated (label entries can be from different data sets). Returns an error code.

c.lastPosition

```
long long c.lastPosition()
```

This function will return the Time or State value of the last position for this correlator depending on the current bias. If the bias setting for *c* is State, then the value found under the State column at the position of the pointer *c* will be returned. If the bias setting for *c* is Time, then the value found under the Time column at the position of the pointer *c* will be returned. Time values are in picoseconds.

c.next

```
int c.next( TDKCorrelatorValue& cv)
```

This function gets the next TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It advances the pointer to the next sample.

c.peekNext

```
int c.peekNext( TDKCorrelatorValue& cv)
```

This function gets the next TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It does not move the pointer.

c.peekPrev

```
int c.peekPrev(TDKCorrelatorValue& cv)
```

This function gets the previous TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It does not change the position of the pointer.

c.prev

```
int c.prev(TDKCorrelatorValue& cv)
```

This function gets the previous TDKCorrelatorValue *cv* in the correlation. Returns 1 for valid data, 0 for invalid data. It moves the pointer back to the previous sample.

c.reset

```
int c.reset()
```

This function resets the position of the pointer to the first position. Returns an error code.

c.resetAtEnd

```
int c.resetAtEnd()
```

This function resets the position of the pointer to the last position. Returns an error code.

c.setPosition

```
int c.setPosition(long long position)
int c.setPosition(TDKDataSet ds, long long position)
```

The setPosition() functions reset the position of the pointer to *position*. In the first form, the position is taken from the reference data set, while in the second form, the position is taken for the TDKDataSet *ds*, which may be any data set involved in the correlation.

c.setStateBias

```
int c.setStateBias()
```

This function sets State bias for the data set. Bias (State or Timing) indicates the type of information the rest of these functions will operate on. This function must be called before the call to initialize(). Returns an error code.

c.setTimeBias

```
int c.setTimeBias()
```

This function sets Timing bias (the default) for the correlator. Bias (State or Timing) indicates the type of information the rest of these functions will operate on. This function must be called before the call to initialize(). Returns an error code.

TDKCorrelatorValue Functions

cv.getData

```
int cv.getData(TDKLabelEntry le, unsigned int &d) //
Integral data
int cv.getData(TDKLabelEntry le, String &d) // Text data
int cv.getData(TDKLabelEntry le, double &d) // Analog
data
```

This function attempts to retrieve the data value for a label entry *le*. If the TDKCorrelatorValue *cv* contains valid information for the label entry *le*, true (1) is returned, otherwise it returns false (0). The data value retrieved is stored in *d*. The function is overloaded for each data type possible (Integral, Text, or Analog).

cv.getState

```
long long cv.getState(TDKLabelEntry le)
```

This function returns the State number of the label entry *le* at the current position of the TDKCorrelatorValue *cv*.

cv.isChanged

```
int cv.isChanged(TDKLabelEntry le)
```

This function returns true if the TDKCorrelatorValue *cv* contains changed information for the given label entry *le*. The actual data found at the label entry position pointed to by *cv* is compared to the previous *cv* label entry data value. If these two values are the same, then this function returns false (0) otherwise it returns true (1).

cv.isHeld

```
int cv.isHeld(TDKLabelEntry le)
```

This function returns true (1) if the TDKCorrelatorValue *cv* contains held information for the given label entry *le* otherwise returns false (0). Often times held data values are discarded.

cv.isValid

```
int cv.isValid(TDKLabelEntry le)
```

This function returns true (1) if the TDKCorrelatorValue *cv* contains valid information for the given label entry *le* otherwise returns false (0).

TDKDataGroup Functions

dg.getDataSetNames

```
int dg.getDataSetNames( StringList& names)
```

This function returns the number of data sets present in the data group and also puts the names into the StringList names that is passed as a parameter. names is reSize()'d to the number of data sets. Use dg.getNumberOfDataSets or StringList function names.length to determine the length of the list. Use StringList function s[n] to retrieve items from the list.

dg.getNumberOfDataSets

```
int dg.getNumberOfDataSets()
```

This function returns the number of data sets present in the data group.

dg.isStateCorrelatable

```
int dg.isStateCorrelatable()
```

This function returns true if the data group is state correlatable.

dg.isTimeCorrelatable

```
int dg.isTimeCorrelatable()
```

This function returns true if the data group is time correlatable.

dg.removeDataSet

```
int dg.removeDataSet(TDKDataSet ds)
```

This function removes the data set to which *ds* is attached. Returns 1 for success, 0 for failure. This function is useful in situations where a new data set is to be displayed without showing the original data set. The original data set can be removed by using this function and passing in the data set reference to the original data set.

dg.setStateCrossCorrelation

```
int dg.setStateCrossCorrelation()
```

This function should be called in a multiple data set situation to tell the system that they should be correlated by state. Returns an error code.

dg.setTimeCrossCorrelation

```
int dg.setTimeCrossCorrelation()
```

This function should be called in a multiple data set situation to tell the system that they should be correlated by time. Returns an error code.

TDKDataSet Functions

ds.attach

```
int ds.attach( TDKDataGroup& dg, String name )
int ds.attach( TDKDataGroup& dg )
```

The first attach function associates the variable *ds* to the first data set in the data group *dg* whose origin or base name is *name*. The second attach function associates the variable *ds* with the first incoming data set. It is permissible to re-attach() the same data set variable to another incoming data set. The effect of this is as if it had never been attach()ed in the first place. Returns an error code.

ds.createState

```
int ds.createState( TDKDataGroup& dg,
String name,
unsigned int len,
unsigned int triggerRow,
long long correlationStateOffset)
```

This function creates a new data set with state information called *name* and with *len* number of samples. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationStateOffset* tells how the data sets trigger position matches up in time in case there is more than one data set. Returns an error code.

ds.createTimePeriodic

```
int ds.createTimePeriodic( TDKDataGroup& dg,
String name,
unsigned int len,
unsigned int triggerRow,
long long correlationTimeOffset,
long long samplePeriod)
```

This function creates a new data set called *name* with time period information and *len* number of samples. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationTimeOffset* tells how the data set trigger position matches up in time in case there is more than one data set. The *samplePeriod* tells the time instants of each of the samples. Returns are error code.

ds.createTimeTags

```
int ds.createTimeTags(TDKDataGroup& dg,
String name,
unsigned int len,
unsigned int triggerRow,
long long correlationTimeOffset,
long long samplePeriod)
```

This function creates a new data set *ds* called *name* with time tags and *len* number of samples. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationTimeOffset* tells how the data set trigger position matches up in time in case there is more than one data set. The *samplePeriod* tells the time instants of each of the samples. Returns an error code.

ds.createTimeTags

```
int ds.createTimeTags(TDKDataGroup& dg,
String name,
TDKDataSet origDS,
unsigned int triggerRow,
long long correlationTimeOffset,
long long samplePeriod)
```

This function creates a new data set *ds* called *name* which is a copy of *origDS* with timeTags added. *ds* will contain all the label entries of *origDS*. The position of the trigger row is placed *triggerRow* number of samples from the first sample. The *correlationTimeOffset* tells how the data set trigger position matches up in time in case there is more than one data set. The *samplePeriod* tells the time instants of each of the samples. Returns an error code.

ds.displayStateNumberLabel

```
ds.displayStateNumberLabel( bool )
```

Disables (or enables) the display of the state number labels for any dataset. If the parameter is set to "false", the state number label for the associated dataset will not appear and will not be available in the listing display. The default value for a dataset is "true" meaning the state number label will appear in the listing display, even though it is not a label that is explicitly created in the TDK code.

ds.filter

```
int ds.filter(long long s)
```

This function will remove the given state *s* as if it had been removed by the Pattern Filter. Returns an error code.

ds.filterAllStates

```
int ds.filterAllStates()
```

This function will remove all states as if the Pattern Filter had removed them. Returns an error code.

ds.firstPosition

```
long long ds.firstPosition()
```

This function will return the Time or State value of the first position for this data set depending on the current bias.

If the bias setting for *ds* is State, then the value found under the State column at the position of the pointer *ds* will be returned. If the bias setting for *ds* is Time, then the value found under the Time column at the position of the pointer *ds* will be returned. Time values are in picoseconds.

ds.getBeginTime

```
int ds.getBeginTime(int A[ ])
```

This function changes the given array *A* of integers to a time representing the approximate start time of the run. Returns an error code. The meanings are as follows:

```
A[0] /* years since 1900 */
A[1] /* month of year - [0,11] */
A[2] /* day of month - [1,31] */
A[3] /* hours - [0,23] */
A[4] /* minutes after the hour - [0,59] */
A[5] /* seconds after the minute - [0,59] */
```

ds.getCorrelationState

long long ds.getCorrelationState()

This function will return the State value of the trigger position for this data set for use in correlating between multiple data sets.

ds.getCorrelationTime

```
long long ds.getCorrelationTime()
```

This function will return the Time value (in picosceconds) of the trigger position for this data set for use in correlating between multiple data sets.

ds.getEndTime

```
int ds.getEndTime(int A[])
```

This function changes the given array *A* of integers to a time representing the approximate end time of the run. Returns an error code. The meanings are as follows:

```
A[0] /* years since 1900 */
A[1] /* month of year - [0,11] */
A[2] /* day of month - [1,31] */
A[3] /* hours - [0,23] */
A[4] /* minutes after the hour - [0,59] */
A[5] /* seconds after the minute - [0,59] */
```

ds.getLabelEntryNames

```
int ds.getLabelEntryNames( StringList names )
```

This function returns the number of label entry *names* present in the data set and fills the given string array with their names.

ds.getName

```
String ds.getName()
```

This function returns the string name of the data set. This name is a colon-separated list of all the tools that feed into the Tool Development Kit Tool.

ds.getNumberOfLabelEntries

```
int ds.getNumberOfLabelEntries()
```

This function returns the number of label entries contained in the data set.

ds.getNumberOfSamples

```
int ds.getNumberOfSamples()
```

Returns the number of samples present in the data set. All label entries contained in this data set have this number of samples as well, by definition.

ds.getPosition

```
long long ds.getPosition()
```

This function will return the Time or State value of the current position for this data set depending on the current bias.

If the bias setting for *ds* is State, then the value found under the State column at the position of the pointer *ds* will be returned. If the bias setting for *ds* is Time, then the value found under the Time column at the position of the pointer *ds* will be returned. Time values are in picoseconds.

ds.getRunID

```
int ds.getRunID()
```

This function returns the run id of the data set. data set ids are guaranteed to be the same if the data sets originated from the same run. This can be useful for group run situations to check whether two data sets originated from the same run.

ds.getTriggerRow

```
unsigned int ds.getTriggerRow()
```

This function returns the relative sample number of the Trigger Row. Note this is not given in terms of a state number, as found under the State label in listing tool. Rather this number is in terms of the number of samples in the trace starting with the first sample (which is zeroth based) counting down to the trigger row sample. This is equal to the number of samples (length of trace) minus the distance in samples from the first sample of the trace to the sample in the trace containing the trigger row.

ds.isAttached

```
int ds.isAttached()
```

This function returns true if the data set has been attached or created successfully on this run.

ds.lastPosition

```
long long ds.lastPosition()
```

This function will return the Time (in picoseconds) or State value of the last position for this data set depending on the current bias.

If the bias setting for *ds* is State, then the value found under the State column at the position of the pointer *ds* will be returned. If the bias setting for *ds* is Time, then the value found under the Time column at the position of the pointer *ds* will be returned. Time values are in picoseconds.

ds.next

```
int ds.next(long long &t)
int ds.next()
```

This function sets the pointer to the next existing x-axis value within the data set and puts this value into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for ds is Time then *t* will contain the Time value in picoseconds. If it cannot return a valid x-axis position, then 0 is given as the return value. Otherwise, the function returns 1.

ds.peekNext

```
int ds.peekNext(long long &t)
```

Without changing the value of the pointer, this function puts the value of the next time or state position into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If there is no valid next position the value 0 is returned. Otherwise,1 is returned.

ds.peekPrev

```
int ds.peekPrev(long long &t)
```

Without changing the value of the pointer, this function puts the value of the previous time or state position into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If there is no valid previous position the value 0 is returned. Otherwise, 1 is returned.

ds.prev

```
int ds.prev(long long &t)
int ds.prev()
```

This function sets the pointer to the previous existing x-axis value within the data set and puts this value into the parameter *t*. If the bias setting for *ds* is State then *t* will contain the State number value. If the bias setting for *ds* is Time then *t* will contain the Time value in picoseconds. If it cannot return a valid x-axis position, then 0 is given as the return value. Otherwise the function returns 1.

ds.removeLabelEntry

```
int ds.removeLabelEntry(TDKLabelEntry le)
```

The label entry variable *le* passed in is removed from the data set. Returns the number of label entries found in *ds* with the same name as *le*. This value should be 1, unless for some reason there is more than one label entry with the same name in the data set.

ds.replaceNext

```
int ds.replaceNext(long long &data)
```

Store the value of *data* in the position pointed to by the pointer, and then increment the pointer by one sample. If the bias setting for *ds* is State, this function does nothing and returns 0. If the bias setting for *ds* is Time, data is written to the Time sample entry. If the function cannot return a valid x-axis position, 0 is given as the return value. Otherwise the function returns 1.

ds.replacePrev

```
int ds.replacePrev(long long &data)
```

Decrement the pointer by one sample, and then store the value of *data* in the position pointed to by the pointer. If the bias setting for ds is State, this function does nothing and returns 0. If the bias setting for *ds* is Time, data is written to the Time sample entry. If the function cannot return a valid x-axis position, 0 is given as the return value. Otherwise the function returns 1.

ds.reset

```
void ds.reset()
```

This function resets the pointer before the first item in the data set. The *ds* pointer will by default point to this state upon creating a new data set or attaching a data set variable to an existing data set.

ds.resetAtEnd

```
void ds.resetAtEnd()
```

This function resets the pointer past the last item in the data set.

ds.setPosition

```
int ds.setPosition(long long t)
```

The current position can be set with this function. Then parameter *t* is interpreted according to the current bias as being Time or State information. This function has no effect on the label entries it contains. Returns an error code.

ds.setStateBias

```
int ds.setStateBias()
```

This function sets State bias for the data set. Bias (state or timing) indicates the type of information many of the data set functions will operate on. The default bias is State, since timing information may not exist. Returns an error code.

ds.setTimeBias

```
int ds.setTimeBias()
```

This function sets Timing bias for the data set. Bias (state or timing) indicates the type of information many of the data set functions will operate on. The default bias is State, since timing information may not exist. Returns an error code.

ds.unfilter

```
int ds.unfilter(long long s)
```

If a state has been filtered using the int ds.filter() function, a call to int ds.unfilter(long long s) makes the state s visible again. This function is not valid for states that have been filtered by the Pattern Filter Tool itself, as the Tool Development Kit tool does not access these states. Returns an error code.

TDKLabelEntry Functions

le.attach

```
int le.attach( TDKDataSet ds, String name )
```

Assigns *le* to the label entry with *name* contained in data set *ds*. The name must match exactly. The data set variable *ds* must be attached before calling this function. Returns an error code.

le.create

```
int le.create( TDKDataSet ds, String name, TDKLabelEntry
orig )
```

This function makes *le* a copy of the TDKLabelEntry *orig*, which is passed as a parameter. This allows the tool to modify the incoming data, if necessary. Note that Bias and Position information are not copied over from *orig* to *le*.

le.createAnalogData

```
int le.createAnalogData( TDKDataSet ds,
String name,
double Offset,
double FullScaleVolts )
```

Used to create a brand new Analog label entry. *name* is the name of the label entry as it appears to downstream tools. *ds* is the data set for which the new label entry is created for. Information is entered into the label entry *le* by using the TDKLabelEntry replace functions.

le.createIntegralData

```
int le.createIntegralData( TDKDataSet ds, String name,
int width )
```

Used to create a brand new Integral label entry. *name* is the name of the label entry as it appears to downstream tools. *ds* is the data set for which the new label entry is created for. *width* is how wide in bits the label entry should be. Information is entered into the label entry *le* by using the TDKLabelEntry replace functions.

le.createTextData

```
int le.createTextData( TDKDataSet ds, String name,
int width )
```

Used to create a brand new Text label entry. *name* is the name of the label entry as it appears to downstream tools. *ds* is the data set for which the new label entry is created for. *width* is how wide in bits the label entry should be. Text label entries can be arbitrarily wide. Information is entered into the label entry *le* by using the TDKLabelEntry replace functions.

le.firstPosition

```
long long le.firstPosition()
```

This function returns the State number or Time, depending on the bias, at the location of the first position. If the bias setting for *le* is State, then the value found under the State column at the position of the pointer *le* will be returned. If the bias setting for *le* is Time, then the value found under the Time column at the position of the pointer *le* will be returned. Time values are in picoseconds.

le.formatBin

```
String le.formatBin(unsigned int val)
```

Converts the value *val* into binary base. Returns a string value that can be appended or included into other strings for output to down-stream tools.

le.formatDec

```
String le.formatDec(unsigned int val)
```

Converts the value *val* into decimal base. Returns a string value that can be appended or included into other strings for output to down-stream tools.

le.formatHex

```
String le.formatHex(unsigned int val)
```

Converts the value *val* into hex base. Returns a string value that can be appended or included into other strings for output to down-stream tools.

le.formatLine

```
String le.formatLine(unsigned int val)
```

This function returns the file name and line number information for a value for use in source correlation.

le.formatOct

```
String le.formatOct(unsigned int val)
```

Converts the value *val* into octal base. Returns a string value that can be appended or included into other strings for output to down-stream tools.

le.formatSymbol

```
String le.formatSymbol(unsigned int val)
```

Lookup *val* in the symbol table of the label entry and return its string symbol value. If it has none, the empty string is returned. See also the system routines formatXXX()s for converting sample values into different bases.

le.formatTwos

```
String le.formatTwos(unsigned int val)
```

Converts the value *val* into Twos base. Returns a string value that can be appended or included into other strings for output to down-stream tools.

le.getName

```
String le.getName()
```

This function returns the name of the label entry.

le.getPosition

```
long long le.getPosition()
```

This function returns the State number or Time, depending on the bias, at the location of the pointer. If the bias setting for *le* is State, then the value found under the State column at the position of the pointer *le* will be returned. If the bias setting for *le* is Time, then the value found under the Time column at the position of the pointer *le* will be returned. Time values are in picoseconds.

le.getWidth

```
int le.getWidth()
```

This function returns the width in bits of the label entry.

le.isAnalogData

```
int le.isAnalogData()
```

This function returns true if the label entry contains analog data.

le.isAttached

```
int le.isAttached()
```

This function returns true if the label entry has been successfully attach()ed or create()ed on this run.

le.isIntegralData

```
int le.isIntegralData()
```

This function returns true if the label entry contains integral data.

le.isTextData

```
int le.isTextData()
```

This function returns true if the label entry contains textual data.

le.lastPosition

```
long long le.lastPosition()
```

This function returns the State number or Time, depending on the bias, at the location of the last position. If the bias setting for *le* is State, then the value found under the State column at the position of the pointer *le* will be returned. If the bias setting for *le* is Time, then the value found under the Time column at the position of the pointer *le* will be returned. Time values are in picoseconds.

le.next

```
int le.next(unsigned int &data) // Integral data
int le.next(String &data)       // Text data
int le.next(double &data)       // Analog data
int le.next(unsigned int &data, long long &pos) //
Integral data
int le.next(String &data, long long &pos) // Text data
int le.next(double &data, long long &pos) // Analog data
int le.next()
```

Fetch the data at the position of the pointer *le* into the variable *data* from the label entry and then increment the pointer by one sample. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds. This function may be called with no arguments to change the current position without fetching the data. Returns 1 if it is a valid sample, otherwise returns 0.

le.peekNext

```
int le.peekNext(unsigned int &data) // Integral data
int le.peekNext(String &data) // Text data
int le.peekNext(double &data) // Analog data
int le.peekNext(unsigned int &data, long long &pos)//
Integral data
int le.peekNext(String &data, long long &pos) // Text
data
int le.peekNext(double &data, long long &pos) // Analog
data
```

Fetch the *data* from the label entry at the position of the pointer *le* into *data* but do not change the position of the pointer. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds. Returns 1 if it is a valid sample, otherwise returns 0.

le.peekPrev

```
int le.peekPrev(unsigned int &data) // Integral data
int le.peekPrev(String &data) // Text data
int le.peekPrev(double &data) // Analog data
int le.peekPrev(unsigned int &data, long long& pos)//
Integral data
int le.peekPrev(String &data, long long& pos) // Text
data
int le.peekPrev(double &data, long long& pos) // Analog
data
```

Fetch the *data* from the label entry at the position previous to the pointer *le* into *data* but do not change the position of the pointer. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds Returns 1 if it is a valid sample, otherwise returns 0.

le.prev

```
int le.prev(unsigned int &data) // Integral data
int le.prev(String &data) // Text data
int le.prev(double &data) // Analog data
int le.prev(unsigned int &data, long long &pos) //
Integral data
int le.prev(String &data, long long &pos) // Text data
int le.prev(double &data, long long &pos) // Analog data
int le.prev()
```

Decrement the pointer by one sample and then fetch the *data* at the position of the pointer into the variable *data* from the label entry. *pos* is a variable which returns the position of the sample, depending on the current bias setting. If the bias is Time, this number will be in picoseconds. This function may be called with no arguments to change the current position without fetching the data. Returns 1 if it is a valid sample, otherwise returns 0.

le.replaceNext

```
int le.replaceNext(unsigned int &data) // Integral data
int le.replaceNext(String &data) // Text data
int le.replaceNext(double &data) // Analog data
```

Store the value of *data* in the position pointed to by the pointer, and then increment the pointer by one sample. Returns 1if it is a valid sample, otherwise returns 0.

le.replacePrev

```
int le.replacePrev(unsigned int &data) // Integral data
int le.replacePrev(String &data) // Text data
int le.replacePrev(double &data) // Analog data
```

Decrement the pointer by one sample, and then store the value of *data* in the position pointed to by the pointer. Returns 1 if it is a valid sample, otherwise returns 0.

le.reset

```
void le.reset()
```

This function resets the pointer before the first item in the label entry.

le.resetAtEnd

```
void le.resetAtEnd()
```

This function resets the pointer past the last item in the label entry.

le.search

```
int le.search(long long& state, String& value, int n,
eSearchMode mode)
```

This function finds the *nth* state from the current position whose data is the same as *value* if *mode* equals *Search_pattern*, and sets state to the position of the match. If *mode* equals *Search_notpattern*, the search finds the *nth* state from the current position whose data is not the same as *value* and sets *state* to the position of the non-match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found. eSearchMode is an enumerated type with the following values:

```
enum eSearchMode
{
   Search_pattern,
   Search_notpattern
}
```

le.searchAndColorAllPattern

```
int le.searchAndColorAllPattern(int color,
unsigned int value, unsigned int mask)
```

This function will display all states in color *color* in the Listing Tool that match *value* and *mask*. This function returns the number of states colored.

le.searchAndColorAllRange

```
int le.searchAndColorAllRange(int color,
unsigned int lo, unsigned int hi)
```

This function will display all states in color *color* in the Listing Tool that are between *lo* and *hi*, inclusive. This function returns the number of states colored.

le.searchAndColorAllNotPattern

```
int le.searchAndColorAllNotPattern(int color,
unsigned int value, unsigned int mask)
```

This function will display all states in color *color* in the Listing Tool that do not match *value* and *mask*. This function returns the number of states colored.

le.searchAndColorAllNotRange

```
int le.searchAndColorAllNotRange(int color,
unsigned int lo, unsigned int hi)
```

This function will display all states in color *color* in the Listing Tool that are not between *lo* and *hi*, inclusive. This function returns the number of states colored.

le.searchAndHighLightAllPattern

```
int le.searchAndHighLightAllPattern(unsigned int value,
unsigned int mask)
```

This function will display all states in a highlighted mode in the Listing Tool that match *value* and *mask*. This function returns the number of states highlighted.

le.searchAndHighLightAllRange

```
int le.searchAndHighLightAllRange(unsigned int lo,
unsigned int hi)
```

This function will display all states in a highlighted mode in the Listing Tool that are between *lo* and *hi*, inclusive. This function returns the number of states highlighted.

le.searchAndHighLightAllNotPattern

```
int le.searchAndHighLightAllNotPattern(unsigned int
value, unsigned int mask)
```

This function will display all states in a highlighted mode in the Listing Tool that do not match *value* and *mask*. This function returns the number of states highlighted.

le.searchAndHighLightAllNotRange

```
int le.searchAndHighLightAllNotRange(unsigned int lo,
unsigned int hi)
```

This function will display all states in a highlighted mode in the Listing Tool that are not between *lo* and *hi*, inclusive. This function returns the number of states highlighted.

le.searchNotPattern

```
int le.searchNotPattern(long long& state, unsigned int
value, unsigned int mask, int n)
```

This function finds the *nth* state from the current position whose data does not match *value* and *mask*, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.searchNotRange

```
int le.searchNotRange(long long& state, unsigned int lo,
unsigned int hi, int n)
```

This function finds the *nth* state from the current position whose data is not between *lo* and *hi*, inclusive, and sets state to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.searchPattern

```
int le.searchPattern(long long& state, unsigned int
value, unsigned int mask, int n)
```

This function finds the *nth* state from the current position whose data matches *value* and *mask*, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.searchRange

```
int le.searchRange(long long& state, unsigned int lo,
unsigned int hi, int n)
```

This function finds the *nth* state from the current position whose data is between *lo* and *hi*, inclusive, and sets *state* to the position of the match. If *n* is negative, the search is in reverse. A return value of 0 means not found, a return value of 1 means found.

le.setColor

```
int le.setColor(long long state, int color)
```

This functions displays the state *state* in color *color* in a downstream Listing Tool. Returns an error code.

le.setHighlight

```
int le.setHighlight(long long state)
```

This function displays the state *state* in a highlighted mode in a downstream Listing Tool. Returns an error code.

le.setName

```
void le.setName( String newName )
```

This function changes the name of the label entry to *newName*.

le.setPosition

```
int le.setPosition(long long s)
```

This function sets the pointer *le* to State number *s* or time *s*, depending on the current bias. If the bias setting is State, then *s* is interpretted as the State number value. If the bias setting is Time, then *s* is interpretted as the Time value in picoseconds. Because some states may have been filtered out, the given state *s* may not exist. In this case, the position will be set to the previous existing state. If no previous state exists, then the position will be set before the first state. Returns an error code.

le.setStateBias

```
int le.setStateBias()
```

This function sets State bias for the label entry. Bias (state or timing) indicates the type of information the iteration functions will operate on. The default bias is State, since Timing in-formation may not exist. Returns an error code.

le.setTimeBias

```
int le.setTimeBias()
```

This function sets Timing bias for the label entry. Bias (state or timing) indicates the type of information the iteration functions will operate on. The default bias is State, since Timing information may not exist. Returns an error code.

Time Functions

microSec

```
long long microSec(double t)
```

This function takes argument *t* that represents a time in microseconds and converts it to a long long type in picoseconds.

milliSec

```
long long milliSec(double t)
```

This function takes argument *t* that represents a time in milliseconds and converts it to a long long type in picoseconds.

nanoSec

```
long long nanoSec(double t)
```

This function takes argument *t* that represents a time in nanoseconds and converts it to a long long type in picoseconds.

picoSec

```
long long picoSec(double t)
```

This function takes argument *t* that represents a time in picoseconds and converts it to a long long type in picoseconds.

sec

```
long long sec(double t)
```

This function takes argument *t* that represents a time in seconds and converts it to a long long type in picoseconds.

timeToString

```
String timeToString(long long t)
```

This function takes argument *t* that represents a time in picoseconds and converts it to a string that is formatted in terms of scientific units similar to the way time is displayed in the lister tool.

Index

Index

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (C) (1) (ii) of the Rights in Technical Data and Computer Software Clause in DFARS 252.227-7013. Agilent Technologies, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A. Rights for non-DOD U.S. Government Departments and Agencies are set forth in FAR 52.227-19 (c) (1,2).

Document Warranty

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

Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose.

Agilent Technologies shall not be liable for errors contained herein or for damages in connection with the furnishing, performance, or use of this material.

Safety

This apparatus has been designed and tested in accordance with IEC Publication 1010, Safety Requirements for Measuring Apparatus, and has been supplied in a safe condition. This is a Safety Class I instrument (provided with terminal for protective earthing). Before applying power, verify that the correct safety precautions are taken (see the following warnings). In addition, note the external markings on the instrument that are described under "Safety Symbols."

Warning

- Before turning on the instrument, you must connect the protective earth terminal of the instrument to the protective conductor of the (mains) power cord. The mains plug shall only be inserted in a socket outlet provided with a protective earth contact. You must not negate the protective action by using an extension cord (power cable) without a protective conductor (grounding). Grounding one conductor of a two-conductor outlet is not sufficient protection.
- Only fuses with the required rated current, voltage, and specified type (normal blow, time delay, etc.) should be used. Do not use repaired fuses or short-circuited fuseholders. To do so could cause a shock of fire hazard.
- Service instructions are for trained service personnel. To avoid dangerous electric shock, do not perform any service unless qualified to do so. Do not attempt internal service or adjustment unless another person, capable of rendering first aid and resuscitation, is present.
- If you energize this instrument by an auto transformer (for voltage reduction), make sure the common terminal is connected to the earth terminal of the power source.
- Whenever it is likely that the ground protection is impaired, you must make the instrument inoperative and secure it against any unintended operation.
- Do not operate the instrument in the presence of flammable gasses or fumes. Operation of any electrical instrument in such an environment constitutes a definite safety hazard.
- Do not install substitute parts or perform any unauthorized modification to the instrument.
- Capacitors inside the instrument may retain a charge even if the instrument is disconnected from its source of supply.

Safety Symbols

⚠

Instruction manual symbol: the product is marked with this symbol when it is necessary for you to refer to the instruction manual in order to protect against damage to the product.

⚡

Hazardous voltage symbol.

⏚

Earth terminal symbol: Used to indicate a circuit common connected to grounded chassis.

WARNING

The Warning sign denotes a hazard. It calls attention to a procedure, practice, or the like, which, if not correctly performed or adhered to, could result in personal injury. Do not proceed beyond a Warning sign until the indicated conditions are fully understood and met.

CAUTION

The Caution sign denotes a hazard. It calls attention to an operating procedure, practice, or the like, which, if not correctly performed or adhered to, could result in damage to or destruction of part or all of the product. Do not proceed beyond a Caution symbol until the indicated conditions are fully understood or met.

Agilent Technologies
P.O. Box 2197
1900 Garden of the Gods Road
Colorado Springs, CO 80901-2197, U.S.A.

Product Warranty

This Agilent Technologies product has a warranty against defects in material and workmanship for a period of one year from date of shipment. During the warranty period, Agilent Technologies will, at its option, either repair or replace products that prove to be defective.

For warranty service or repair, this product must be returned to a service facility designated by Agilent Technologies.

For products returned to Agilent Technologies for warranty service, the Buyer shall prepay shipping charges to Agilent Technologies and Agilent Technologies shall pay shipping charges to return the product to the Buyer. However, the Buyer shall pay all shipping charges, duties, and taxes for products returned to Agilent Technologies from another country.

Agilent Technologies warrants that its software and firmware designated by Agilent Technologies for use with an instrument will execute its programming instructions when properly installed on that instrument. Agilent Technologies does not warrant that the operation of the instrument software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by the Buyer, Buyer- supplied software or interfacing, unauthorized modification or misuse, operation outside of the environmental specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. Agilent Technologies specifically disclaims the implied warranties of merchantability or fitness for a particular purpose.

Exclusive Remedies

The remedies provided herein are the buyer's sole and exclusive remedies. Agilent Technologies shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Assistance

Product maintenance agreements and other customer assistance agreements are available for Agilent Technologies products. For any assistance, contact your nearest Agilent Technologies Sales Office.

Certification

Agilent Technologies certifies that this product met its published specifications at the time of shipment from the factory. Agilent Technologies further certifies that its calibration measurements are traceable to the United States National Institute of Standards and Technology, to the extent allowed by the Institute's calibration facility, and to the calibration facilities of other International Standards Organization members.

About this edition

This is the *Tool Development Kit User's Guide*.

Publication number
B4605-97006, October 2002
Printed in USA.

Many product updates do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.