Tecplot®

Getting Results

With Tecplot’s Add-on Developer’s Kit

Version 10 Release 3

Tecplot, Inc.
Bellevue, Washington
June 8, 2004

Table of Contents

CHAPTER 1 About Add-ons

INTRODUCTION

This manual describes strategies for creating Tecplot add-ons. Add-ons are executable modules that extend Tecplot's basic functionality. Add-ons are implemented as compiled function libraries, called variously "shared objects," "shared libraries," or "dynamic-link libraries" (DLLs). Using the Tecplot application programming interface you can create add-ons to generate plots, load data from files, manipulate or analyze data, or perform a broad variety of specialized tasks. Because add-ons are shared objects, you do not need to link them into Tecplot. You are not limited to using the compilers Tecplot uses, nor do you have to compile, or recompile, large libraries of Tecplot function calls.

Different operating systems have different ways of creating and using shared libraries. Add-on Developer's Kit (ADK) provides utilities that mask most of these differences for related programs. All Windows or UNIX systems will behave in a similar fashion. ADK tools will resolve the differences for you. All of the examples of the source code shown in this manual are included in the Tecplot distribution and are found in the **`adk/samples`** sub-directory below the Tecplot Home Directory. To read more about advanced topics, see the *Add-on Developer's Kit User's Manual* and the *Add-on Developer's Kit On-line Reference*. All are included as Adobe PDF files with your Tecplot distribution or are available at **www.tecplot.com/support/tecplot_documentation.htm.**

CHAPTER 2 Creating Add-ons under Windows

SETTING UP TO BUILD ADD-ONS UNDER WINDOWS

To set up to build add-ons, install Tecplot Version 10. Make sure the Tecplot Add-on Developers Kit option was selected during installation. To verify that the Add-on Developer's Kit was installed on your computer, look in your Tecplot Home Directory for the **`ADK`** sub-directory. If the **`ADK`** sub-directory is not present, you will need to re-install Tecplot Version 10.

If you plan on using Tecplot GUI Builder (TGB), make sure the following line is in the **`tecplot.add`** file in the Tecplot Home Directory:

```
$!LoadAddon "guibld"
```

Tecplot GUI Builder is discussed in detail in Section 2.3, as well as in the Tecplot Add-On Developers Kit Users Manual.

CREATING AN ADD-ON WITH VISUAL C++

Tecplot Add-on Wizard is included in the Tecplot installation and is fully integrated with Visual C++ Version 5.0 or higher. To begin, select **New** from Visual C++'s File menu, then click on the **Projects** tab. For the project type select "Tecplot 10 Add-on Wizard" and follow the prompts. Since Tecplot add-ons are DLLs, Tecplot Add-on Wizard will automatically create a DLL workspace, set the proper link libraries, include paths, and generate default source code files.

Note: FORTRAN under Windows is only supported if you are using Compaq Visual FORTRAN.

After running Tecplot Add-on Wizard, you must complete the following steps:

1. Select **Settings** from the Visual C++ Project menu.
2. Click on **Debug**.
3. Select **General**.
4. Set Executable for debug session to be **`tecplot.exe`** (including the full path if necessary).
5. Set the working directory to Debug.
6. Set the program arguments to be **`projectname.dll`**. *Projectname* is the base name of your DLL.

Compiling and debugging your add-on is now a matter of using the Developer's Studio environment as you would for any other DLL project.

NOTE: If you find that when you try and run your add-on, you instead run a pre-existing add-on with the same name that exists in your path, try the following settings instead

1. Set the Working directory to empty;

2. Set the Program arguments to be **`Debug/projectname.dll`**

See Chapter 6, "Running Tecplot with Add-ons (UNIX and Windows)," in the *ADK User's Manual* for detailed instructions on loading add-ons.

DIALOG CREATION WITH TECPLOT GUI BUILDER

Tecplot Add-on Developer's Kit includes a graphical user interface (GUI) builder called Tecplot GUI Builder (TGB). TGB is provided in the Tecplot distribution. The Tecplot ADK User's Manual outlines its use. When you run Tecplot Add-on Wizard from Developer Studio, a default set of TGB files are created. This default code will display a blank dialog, which may be modal or modeless. These project settings are made automatically by Tecplot Add-on Wizard. You will edit a TGB dialog layout in Tecplot using TGB add-on, since TGB dialog layouts are stored as Tecplot layout files. Developer Studio is not involved in editing or maintaining TGB dialog layouts.

CHAPTER 3 Creating Add-ons under UNIX

SETTING UP TO BUILD ADD-ONS

To create Tecplot add-ons under Unix, you must set up a working directory where source code can be created and edited. This directory will hereafter be called the Add-on Development Root Directory. You may create any number of add-ons in the Add-on Development Root Directory.

To set up for building add-ons do the following:

1. Install Tecplot if you have not done so already. Make sure the Add-on Development Tools option was selected during the installation process.

2. Create the Add-on Development Root Directory if you have not done so already. This can be anywhere you choose.

3. Be sure that you have the **`TEC100HOME`** environment variable defined and assigned to the directory where Tecplot was installed.

4. Be sure your **`PATH`** environment variable includes the following:

```
$TEC100HOME/bin:$TEC100HOME/adk/bin
```

5. Create a new file called **`tecdev.add`** in the directory created in step 2 (i.e. your Add-on Development Root Directory). Edit the file and add the following line:

```
#!MC 1000
```

6. (Optional) If you plan on using the Tecplot GUI builder, then add the following line to the **`tecdev.add`** file in your Add-on Development Root Directory:

```
$!LoadAddon "|TECHOME|/lib/libguibld"
```

7. Set the environment variable **`TECADDONDEVDIR`** to the path of the directory created in step 2.

8. Set the environment variable **`TECADDONDEVPLATFORM`** to one of the following:

```
hp7xx.11       linuxg23.24  linux.24      macx.101
hp7xx64.65     linuxi64.24  linux64.24     ibmx.43
sgix.65        sgix64.65    sun464.57      sun4.57
```

From this point on, when you want to test the add-ons you are developing, use the **`-develop`** flag when running Tecplot. Later when you want to make your add-on accessible to all who run Tecplot, just copy the shared object library to the **`lib`** subdirectory below the Tecplot Home Directory and include the command: **`$!LoadAddOn"|TECHOME|/lib/lib`***MyAddOnName***`"`** in the tecplot.add file in the Tecplot Home Directory.

CREATING A NEW ADD-ON

1. Go to the Add-on Development Root Directory (i.e., the directory created in step 2 of Section , “Setting Up to Build Add-ons.”).

2. Type: **`CreateNewAddOn`**

This will ask you a few questions about the add-on to be built, including whether or not you intend to use the Tecplot GUI Builder. When this is finished, you will have a new sub-directory named MyAddOnName, where MyAddOnName is the name that you supplied in step 2 while running CreateNewAddOn. This subdirectory contains a set of file. These files can be compiled to create a minimal add-on.

1. Edit the **`tecdev.add`** file located in the Add-on Development Root directory and add the following line: **`$!LoadAddOn "|$TECADDONDEVDIR|/lib`***MyAddOnName*" where MyAddOnName is the name you supplied in step 2 while running CreateNewAddOn.

For your add-on to communicate with Tecplot it must do the following:

- Make public an “initialization” function named **`InitTecAddOn.`** When you run **`CreateNewAddOn`** this function is created automatically for you and is located in the file **`main.c`** (or **`main.cpp`**). When Tecplot starts up it scans the **`tecdev.add`** file, loads named shared object libraries and makes a call to the **`InitTecAddOn`** function. The initialization function typically includes a call to add a converter, add a loader, register a curve-fit, or add an item to the Tools menu, so the add-on can be accessed from the Tecplot interface.
- Make calls to the TecUtil functions available from the ADK via the **`libtec`** shared object library. These functions allow you to do a wider range of tasks than can be done through the Tecplot interface itself.
- If your add-on does not require a custom built GUI, you will, at this point, have a source file named **`main.c`**, and perhaps a source file named **`engine.c`**. The latter file contains callback functions for data loaders, data convertors, or curve-fits.

CREATING THE GRAPHICAL USER INTERFACE FOR YOUR ADD-ON

The Tecplot Add-on Developers Kit includes a simple GUI builder called Tecplot GUI Builder (TGB). You are not restricted to this GUI builder. You may use a commercial GUI builder like *Builder Xcessory* or *X-Designer*. Chapter 8 of this document outlines how to use the Tecplot GUI Builder. It is provided on the Tecplot CD. When you run **`CreateNewAddOn`** and choose to use the TGB, a starter set of TGB files are created for you.

COMPILING THE ADD-ON

Using Runmake

If you used **`CreateNewAddOn`**, compiling the add-on is straightforward. Go to the subdirectory where your add-on source code is located and type **`Runmake.`** You will be prompted for the platform type and the type of executable to create.

If you know the platform name and the build option ahead of time then you can run **`Runmake`** without the questions. For example, to compile on an SGI machine under IRIX 6.5 and create a debug version use:

```
Runmake sgix.65 -debug
```

To make a release version use: **`Runmake sgix.65 -release`**

If all goes well with the compile, you will end up with a shared object library located in **`../lib/`***platform/buildtype*. Running Tecplot with the **`-develop`** flag automatically directs it to look for your library in this directory.

Note: If the Tecplot Home Directory and your Add-on Development Directory are located in directories that can be remotely mounted by other UNIX computers, then you can log on to those computers and use **`Runmake`** as described earlier. The resulting shared library will be stored in the appropriate subdirectory for the computer platform.

Editing the CustomMake File

The **`Runmake`** command used to build your add-on actually invokes the UNIX **`make`** program with a large list of flags that customize the make process for your platform. Just prior to calling **`make`**, the **`Runmake`** shell script checks to see if a local file called **`CustomMake`** exists and is executable. If so, it runs the **`CustomMake`** shell script in place and then runs **`make`**. This process allows you to add to or completely replace any assignments made by **`Runmake`**.

For example, suppose you want to add an additional flag called **`-xg`** to the **`cc`** compile command. You could do so by editing the local **`CustomMake`** shell script in the sub-directory of your add-on and adding:

```
        CFLAGS="$CFLAGS -xg"
```

This replaces **`CFLAGS`** (i.e. the flags used with the **`cc`** command) with its old contents plus the **`-xg`** flag.

The default **`CustomMake`** file created in your add-on directory when you run **`CreateNewAddOn`** contains edit instructions including an explanation of the flags available for you to change.

CHAPTER 4   Hello World!

INTRODUCTION TO THE HELLO WORLD ADD-ON

Hello World, the Tecplot add-on you will create in this chapter, is an example of how an add-on performs tasks or functions for you. *Hello World* will appear under Tecplot's Tools menu as "Hello World!". When selected, a dialog displaying the text "Hello World!" will appear. To create this add-on you should have first read Chapter 2 "Creating Add-ons under Windows," or Chapter 3 "Creating Add-ons under UNIX " All of the code presented in this chapter is platform independent, allowing you to work in either a UNIX or Windows environment. All of the example source code shown in this manual is included in the Tecplot distribution and is found in the **`adk/samples`** sub-directory below the Tecplot Home Directory. *Hello World* uses source code files created by the **`CreateNewAddOn`** script (UNIX), or Tecplot Add-on Wizard (Windows). Our project name will be "hiwrld" and the add-on name will be "Hello World."

When running **`CreateNewAddOn`** or Tecplot Add-on Wizard, answer the questions as follows:

• Project name (base name):	hiwrld
• Add-on name:	Hello World
• Company name:	[*Your Company Name*]
• Type of Add-on	General Purpose
• Language:	C
• Use TGB to create a platform-independent GUI?:	No
• Add a menu callback to the Tecplot "Tools" menu?:	Yes
• Menu Text:	Hello World!

The question "Use TGB to create a platform-independent GUI" option specifies that you will use Tecplot GUI Builder in your add-on. After running the **`CreateNewAddOn`** script or Tecplot Add-on Wizard you should have the following files: **`ADDGLBL.h`** and **`main.c.`**You will have other files specific to your platform, but only those above will be modified. Verify that you can compile your add-on project and load it into Tecplot. For UNIX this is done by running the **`Runmake`** script. In Windows, click *Build/Build hiworld.dll*. For detailed information on compiling refer to Chapter 2 "Creating Add-ons under Windows," or Chapter 3 "Creating Add-ons under UNIX." Once you have compiled *Hello World* you can run Tecplot and select "*Hello World!*" from Tecplot's Tools menu. Text is written to standard out (or the debug window in Developer Studio) reading *Menu function called*. When finished, this will read "*Hello World!*" in a dialog.

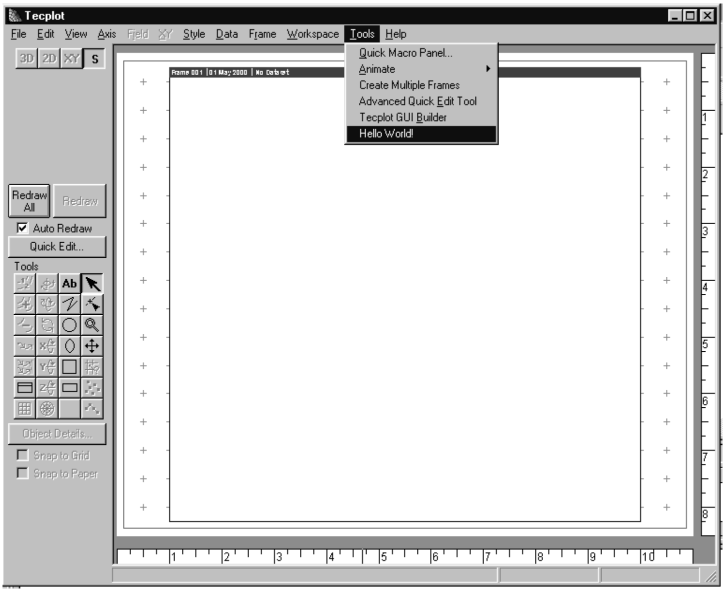
MODIFYING THE MENUCALLBACK() FUNCTION

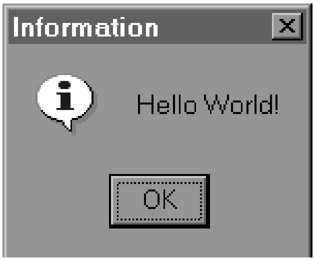
Most add-ons contain a callback function named **`MenuCallback()`**. This is called by Tecplot when the user selects the add-ons registered menu option from the Tools menu. This callback function is registered by the **`TecUtilMenuAddOption()`** function, which is in **`InitTecAddOn().`** These will be discussed in detail later. Because the add-on dialog displays a different message than “Hello World!” it must be edited. New or modified source code is displayed in the bulleted lines. If you are working along with this tutorial, add the bulleted lines only. All **`TecUtil`** functions are explained in the *ADK Reference Manual*.

In **`main.c`**, edit the **`MenuCallBack()`** function as follows:

```
static void STDCALL MenuCallback(void)
{
  TecUtilLockStart(AddOnID):
• TecUtilDialogMessageBox(“Hello World!”,MessageBox_Information);
  TecUtilLockFinish(AddOnID);
}
```

Hello World is now complete. Recompile and run Tecplot.





Hello World!

CHAPTER 5 The Equate Add-on

INTRODUCTION TO THE EQUATE ADD-ON

Equate, the Tecplot add-on you will create in this chapter, is an example of how to query and set field data in an add-on. It will appear on the Tools menu as *Equate*. This add-on multiplies each data point of the first variable in the first zone by a value entered in a dialog text field. All of the examples of the source code shown in this manual are included in the Tecplot distribution and are found in the **`adk/samples`** subdirectory below the Tecplot home directory. *Equate* uses source code files created by the **`CreateNewAddOn`** script (UNIX), or Tecplot Add-on Wizard (Windows). Our project and add-on names will be *Equate*.

When running **`CreateNewAddOn`** or Tecplot Add-on Wizard answer the questions as follows:

• Project name (base name):	Equate
• Add-on name:	Equate
• Company name:	[*Your Company Name*]
• Type of Add-on	General Purpose
• Language:	C
• Use TGB to create a platform-independent GUI?:	Yes
• Add a menu callback to the Tecplot "Tools" menu?:	Yes
• Menu Text:	Equate
• Menu Callback Option:	Launch a modeless dialog
• Dialog title:	Equate

After running the **`CreateNewAddOn`** script, or Tecplot Add-on Wizard you should have the following files:

```
ADDGLBL.h      guicb.c     guibld.c      guidefs.c
GUIDEFS.h      main.c      gui.lay
```

You will have other files specific to your platform, but only those above will be modified. Verify that you can compile your add-on project and load it into Tecplot. For UNIX this is done by running the **`Runmake`** script. In Windows, click on the Tool button. For detailed information on compiling refer to Chapter 2 *Creating Add-ons under Windows* or Chapter 3 *Creating Add-ons under UNIX*.

Now create your main dialog. This will be displayed when *Equate* is selected from Tecplot's Tools menu. The dialog will be modeless with a text field, label, and button. When a user enters a numeric value in the text field and clicks the button, *Equate* will multiply each data point of the first variable in the first zone by that value. Before beginning, be sure that Tecplot GUI Builder (TGB) is available from Tecplot's Tools menu. If TGB is not available, do the following

For Windows:

In the Tecplot Home Directory edit the file **`tecplot.add`** and add the line:
`$!LoadAddOn "guibld"`

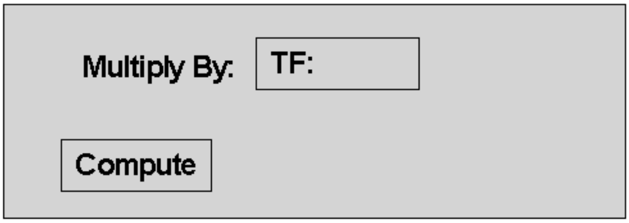
For UNIX:

Edit the file **`tecdev.add`** in your Add-on Development Root Directory and add the line:
`$!LoadAddOn "guibld"`

To create the main dialog, perform the following steps:

1. Run Tecplot and load the gui.lay file for your project. Select Tecplot GUI Builder (TGB) from the Tecplot Tools menu.

2. Resize the frame and edit the layout as follows:



You can edit a control by double-clicking on it and editing as you would text.

NOTE: Although the text fields and buttons are referred to as controls (since they exist in a Tecplot layout file), they are represented by Tecplot text field objects.

3. So that TGB will create meaningful variable names for the text field controls, change their properties in Tecplot. Double-click on the text field "TF:," then select Options.

Note: Do not alter the text string "TF" Tecplot uses this string to identify this control as a Text Field.

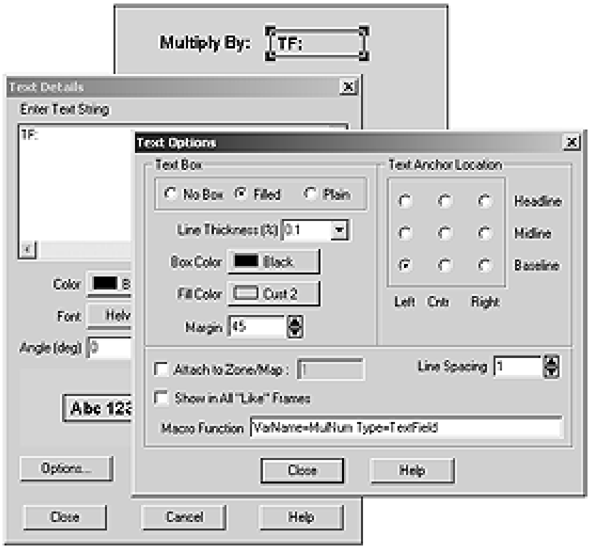
In the Macro Function text field, set VarName=MulNum. This will be the base name of the "Multiply By" callback function which will be named **MulNum_TF_D1_CB**.TGB takes the base name and decorates it with the dialog number, control type and CB (for callback).

4. Double-click on the Compute button, then select Options. Set **`VarName=Compute`** in the "Macro Function" field. The Compute buttoncallback function will be named `Compute_BTN_D1_CB`. Double-click on the "Multiply By" label, then select Options. Set **`VarName=MultiplyBy`** in the "Macro

Function” field.. Callback functions are not generated for labels, but a variable name will be generated and will be named **MultiplyBy_LBL_D1**.

5. In TGB, the title of the dialog is specified in the Edit Current Frame dialog. Double-click on the dialog frame and verify that the Frame Name has been set to “Equate”:

```
ID=1 MODE=MODELESS TITLE=”Equate” OKCLOSEBUTTON=T HELPBUTTON=T
```


Multiply By: TF:

Text Details

Enter Text String: TF:

Color: B

Font: Helv

Angle (deg): 0

Text Options

Text Box: ○ No Box ● Filled ○ Plain

Line Thickness (%): 0.1

Box Color: Black

Fill Color: Cust 2

Margin: 45

Text Anchor Location: ● Left Baseline

□ Attach to Zone/Map: 1

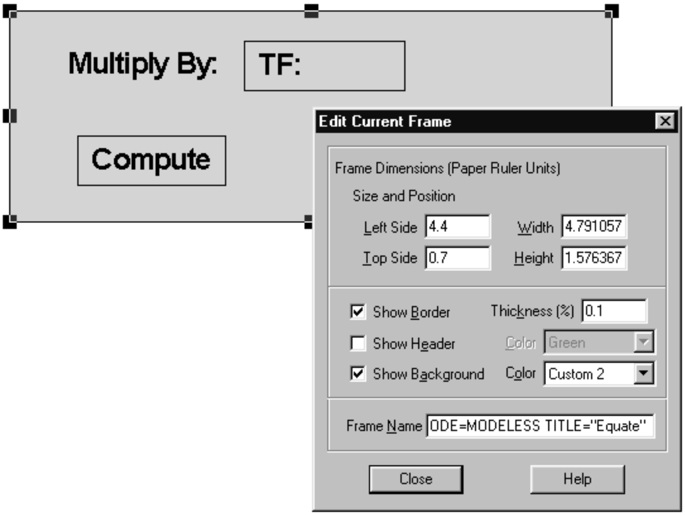
Line Spacing: 1

□ Show in All "Like" Frames

Macro Function: VarName=MulNum Type=TextField

6. You can now build the source for this layout. From the TGB dialog click **Go Build**. If you wish to preview what your dialog will look like when run, click **Preview Layout** from TGB.

7. Rename the file **`guicb.tmp`** to be **`guicb.c`**, replacing the original **`guicb.c`** then compile the source code.


Multiply By: TF:

Edit Current Frame

Frame Dimensions (Paper Ruler Units)

Size and Position

Left Side: 4.4 Width: 4.791057

Top Side: 0.7 Height: 1.576367

☑ Show Border Thickness (%): 0.1

□ Show Header Color: Green

☑ Show Background Color: Custom 2

Frame Name: ODE=MODELESS TITLE="Equate"

GUI SOURCE CODE

Now we will examine the source code files generated by TGB.

- **Files `guidefs.c, GUIDEFS.h:`** Contain the variable names of all of the controls added to the dialog. TGB has taken the variable names specified in the Macro Function Command field and decorated them as follows:

```
int Dialog1Manager    = BADDIALOGID;
int MulNum_TF_D1      = BADDIALOGID;
int Compute_BTN_D1    = BADDIALOGID;
int MultiplyBy_LBL_D1 = BADDIALOGID;
```

- **`TF`** is text field and `D`*n* is the dialog number. Since there is only one dialog box, *n* is 1. For example, the name **`MulNum_TF_D1`** can be decoded as "This variable represents the **`MulNum Text Field`** in Dialog **1**." The variables are initialized to **`BADDIALOGID`** to ensure that they cannot be passed as parameters to any Tec`GUI` library function until the dialog has actually been created. At that time they will be assigned a valid identification.

- **Note:** Never edit these files directly. TGB will generate them every time you click Go Build.

- **File `guibld.c:`** Contains the code used to build the dialogs.

- **Note:** Never edit this file directly. TGB will generate this file every time you click Go Build, so any changes you make will be overwritten. Also, this file is never included directly in the project. Instead, the text of this source code file is included directly in **`guicb.c`** with a **`#include 'guibld.c'`** preprocessor statement at the end of **`guicb.c`**.

- **File `guicb.tmp:`** Contains all of the callbacks for the dialog controls. A callback function is a function you define which is called by Tecplot when an event occurs for a control. For example, a button control will have a callback function for the button pressed event.

- Initially, TGB will generate empty callbacks, but instead of writing them to **`guicb.c`**, it will write them to a file named **`guicb.tmp`**. The reason for this is that TGB does not want to overwrite any code that you may have added to **`guicb.c`**. Thus, whenever you add new controls, you must cut-and-paste the new callback functions in **`guicb.tmp`** into **`guicb.c`**. Note that in Step 7 we copied **`guicb.tmp`** to **`guicb.c`**. This is what you want to do when you first start the project since at that time there is no custom code in **`guicb.c`**.

To see the new dialog:

1. Compile the add-on and run Tecplot.

2. Select *Equate* from the Tools menu.

SETTING UP STATE VARIABLES AND INITIALIZING THE DIALOG FIELDS

When the dialog is first displayed, we need to be sure that the **`MulNum`** text field has a reasonable default value. To avoid using a global variable for **`MulNum`**, the value will be read from the text field and passed to a function called **`Compute()`**. The text field will then be initialized in the **`Dialog1Init_CB()`** function.

Note the following line in **`guicb.c`**:

```
/* This is a string because it is put in a dialog text field */
#define DEFAULT_MULNUM "2"
```

Find the following segment of code in **guicb.c** and note the line beginning with TecGUITextField...

```
static void Dialog1Init_CB(void)
{
  TecUtilLockStart(AddOnID);

  /*<<< Add init code (if necessary) here>>>*/
  TecGUITextFieldSetString(MulNum_TF_D1,DEFAULT_MULNUM);
  TecUtilLockFinish(AddOnID);
}
```

We have defined the default value to be a string, since that is what **TecGUITextFieldSetString()** expects. The **Compute()** function will be called when you click Compute. The function will be prototyped as follows. Note the function call to **Compute()** in **guicb.c**. This function will be written below. Before calling this function, check that a data set is available. If there is, then it is implied that at least one zone and one variable exist.

Edit **guicb.c** as follows:

```
static void Compute_BTN_D1_CB(void)
{
  char *strMulNum = NULL;

  TecUtilLockStart(AddOnID);
  strMulNum = TecGUITextFieldGetString(MulNum_TF_D1);

  if (TecUtilDataSetIsAvailable())
    {
      Compute(atof(strMulNum));
    }
  else
    TecUtilDialogErrMsg("No data set available.");

  TecUtilStringDealloc(&strMulNum);

  TecUtilLockFinish(AddOnID);
}
```

No error checking is done on the input string. As an exercise, use **TecGUITextFieldGetDouble**, **TecGUITextFieldSetDouble**, and **TecGUITextFieldValidateDouble** to do error checking for you.

WRITING THE COMPUTE() FUNCTION

The final task is to write the **Compute()** function. This will multiply each data point of the first variable in the first zone by the input parameter, then send a message to Tecplot that the data set has changed. The recommended way for an add-on to get and set field data is with **FieldData_pa** handles. See the *ADK User's Manual* for a complete discussion of getting and setting data values within Tecplot. Examine **main.c** and note the following function:

```
void Compute(double MulNum)
{
  LgIndex_t IMax;
  LgIndex_t JMax;
  LgIndex_t KMax;
  LgIndex_t i;
  LgIndex_t MaxIndex;
  FieldData_pa FD;
  double Value;
  Set_pa set;

  TecUtilLockStart(AddOnID);

  /* Get the number of data points */
  TecUtilZoneGetInfo(1,     /* Zone */
                     &IMax,
                     &JMax,
                     &KMax,
                     NULL,  /* XVar */
                     NULL,  /* YVar */
                     NULL,  /* ZVar */
                     NULL,  /* NMap */
                     NULL,  /* UVar */
                     NULL,  /* VVar */
                     NULL,  /* WVar */
                     NULL,  /* BVar */
                     NULL,  /* CVar */
                     NULL); /* SVar */

  MaxIndex = IMax * JMax * KMax;

  FD = TecUtilDataValueGetRef(1,1);

  for (i = 1; i <= MaxIndex; i++)
    {
      /* Get the value */
      Value = TecUtilDataValueGetByRef(FD,i);

      /* Change it */
      Value *= MulNum;

      /* And set it back */
      TecUtilDataValueSetByRef(FD,i,Value);
    }

  /* Inform Tecplot that we've changed the data */
  set = TecUtilSetAlloc(FALSE);
  TecUtilSetAddMember(set,1,FALSE); /* Zone 1 */
  TecUtilStateChanged(StateChange_VarsAltered,
                      (ArbParam_t)set);
  TecUtilSetDealloc(&set);

  TecUtilLockFinish(AddOnID);
}
```

Equate is now complete. Recompile and load it into Tecplot. Note that this example add-on is only valid for ordered data as we computed MaxIndex by simply multiplying the dimensions together.

EXERCISES

1. Currently, there is no error checking done on the value entered in the text field. You could enter “ABCDEFG” and **`atof()`** would convert it into 0.0. This could be fixed by adding error checking to the button callback. Use **`TecGUITextFieldValidateDouble`** and **`TecGUITextFieldGetDouble`** for better error checking.

2. Add a multi-selection list box which allows you to select one or more zones from a set.

3. Add a multi-selection list box to select one or more variables from a set.

4. This add-on assumes variable 1 and Zone 1 are “Enabled,” which may not be the case. Add error checking to make sure Zone 1 is enabled (**`TecUtilZoneIsEnabled`**) and variable 1 is enabled (**`TecUtilVarIsEnabled`**).

CHAPTER 6 Extending the Equate Add-on

GETTING STARTED

Now we will examine code that allows *Equate's* compute function to be run from a macro command. All of the examples of the source code shown in this manual are included in the Tecplot distribution. They may be found in: **`TEC100HOME/ADK/Samples`**

EDITING EQUATE

The first step will be to decide what information is required by the add-on. *Equate* only requires that the value is sent to the **`Compute()`** function. To write out the macro command, we will use the **`TecUtilMacroRecordAddOnCommand()`** function. All **`TecUtil`** functions are defined in the *ADK Reference Manual*.

Note the **`Compute_BTN_D1_CB()`** function in **`guicb.c`**:

```
static void Compute_BTN_D1_CB(void)
{
  char *strMulNum = NULL;

  TecUtilLockStart(AddOnID);
  strMulNum = TecGUITextFieldGetString(MulNum_TF_D1);

  if (TecUtilDataSetIsAvailable())
    {
      Compute(atof(strMulNum));
      if (TecUtilMacroIsRecordingActive())
        TecUtilMacroRecordAddOnCommand("equate", strMulNum);
    }
  else
    TecUtilDialogErrMsg("No data set available.");

  TecUtilStringDealloc(&strMulNum);

  TecUtilLockFinish(AddOnID);
}
```

We check to see if a macro is being recorded before we write out the macro command. When **`TecUtilMacroRecordAddOnCommand()`** is called, it will add a line to the macro file that will appear as follows:

```
$!ADDONCOMMAND
ADDONID='equate'
```

```
COMMAND=’2’
```

ADDONID tells Tecplot which add-on to send the command to, and **COMMAND** is the value in the text field of *Equate’s* dialog. Now that a macro command is being written out, write a function to decode it. When a macro is running, Tecplot will send the information following **COMMAND** to the add-on. In this case, the only item that **COMMAND** contains is a number. Tecplot sends all the information following **COMMAND** as a string.

Examine the following function in **main.c**:

```
Boolean_t STDCALL ProcessEquateCommand(char  *CommandString,
                                       char **ErrMsg)
{
  Boolean_t IsOk;

  TecUtilLockStart(AddOnID);

  IsOk = TecUtilDataSetIsAvailable();
  if (IsOk)
    {
      Compute(atof(CommandString));
    }
  else
    {
      *ErrMsg = TecUtilStringAlloc(2000, “Error message”);
      strcpy(*ErrMsg, “No data set available.”);
    }

  TecUtilLockFinish(AddOnID);

  return IsOk;
}
```

Functions that process macro commands may have any name you choose, however, they must have the parameters shown above. This function mirrors, to a certain extent, the **Compute_BTN_D1_CB()** function in **guicb.c**. There is no error checking of the value of **CommandString**. This is left as an exercise. In order to process macros, you must register a callback function. Note that the second parameter of **TecUtilMacroAddCommandCallback()** is the same as the name of our macro processing function.

In **main.c** note the registration of the **ProcessEquateCommand()** macro command callback from within the add-on initialization code:

```
EXPORTFROMADDON void STDCALL InitTecAddOn(void)
{
  TecUtilLockOn();

  AddOnID = TecUtilAddOnRegister(
              100,
              ADDON_NAME,
              “V”ADDON_VERSION”(“TecVersionId”) “ADDON_DATE,
              “Joe Coder”);

  if (TecUtilGetTecplotVersion() < MinTecplotVersionAllowed)
```

```
    {
      char buffer[256];
      sprintf(buffer, “Add-on \”%s\” requires Tecplot “
                      “version %s or greater.”,
              ADDON_NAME, TecVersionId);
      TecUtilDialogErrMsg(buffer);
    }
  else
    {
      InitTGB();

      TecUtilMenuAddOption(“Tools”,
                           “Equate”,
                           ‘E’,
                           MenuCB);
      TecUtilMacroAddCommandCallback(“equate”, ProcessEquateCommand);
    }

  TecUtilLockOff();
}
```

Equate is now complete. Compile and run your add-on. Try recording and playing back various macros to verify that the new functions you have added work properly.

CHAPTER 7 Adding Help

INTRODUCTION

Once your add-on is complete, you will find that there are many details and instructions you would like to make available to users. Online Help is an effective way to include necessary details and instructions. It is the best way to ensure that needed information can be easily accessed from your add-on.

Note: The Help mechanism described in this chapter requires that you have a Web browser available on your platform.

CREATING HELP

Call **`TecUtilHelp`** to launch a help file. Help can be called anywhere within your add-on, although the typical procedure is to start from a dialog's **`HelpButton`** callback. To add help to the *Equate* add-on, create a simple HTML document to serve as your help file, naming it **`equate.html`**.

In **`guicb.c`**, note the call to launch the help in **`Dialog1HelpButton_CB()`**:

```
static void Dialog1HelpButton_CB(void)
{
  TecUtilLockStart(AddOnID);
  TecUtilHelp(“equate.html”,FALSE,0);
  TecUtilLockFinish(AddOnID);
}
```

Place **`equate.html`** in the **`help`** sub-directory below the Tecplot Home Directory, then recompile and reload your add-on. When you click Help on the *Equate* dialog, or press F1, **`equate.html`** will be launched in the default browser.

CHAPTER 8 Creating a Data Converter

CONVERTERS VERSUS LOADERS

Data can be imported into Tecplot using *converter* or *loader* add-ons. A *converter* is used when simple proprietary data files need to be read into Tecplot and it is not necessary to use complex options to decide which portions of the data should be loaded. Converters are simple to create but not as versatile as loaders. A loader displays its own custom dialog for the user to enter the parameters needed to load the data: file name, skip values, and so forth. With converters, Tecplot controls the user interface used to prompt the user for the names of the files to load.

How do Converters work in Tecplot?

A data *converter* is a special type of add-on which can read data in a custom file format and import it into Tecplot. It does this by reading the data and writing out a temporary binary data file. Tecplot loads this temporary file and then discards it. Tecplot queries the user for a file name, then passes it to the converter. If you need to query users for information other than file names, you must use a data *loader*. (Data loaders are discussed in the following chapter.) Given the file name, the procedure used by a converter to import that data is similar to creating a Tecplot binary file using the **`TecIO`** functions. (See the *Tecplot Reference Manual* for more information on using the **`TecIO`** library.)

INTRODUCTION TO THE CONVERTER ADD-ON

Converter, the add-on created in this tutorial, is an example of how to load a comma- or space-delimited list of values into Tecplot. *Converter* will appear under the Import option of Tecplot's File menu. All of the examples of the source code shown in this manual are included in the Tecplot distribution and are found in the **`adk/samples`** sub-directory below the Tecplot home directory. *Converter* uses source code files created by the **`CreateNewAddOn`** script (UNIX), or Tecplot Add-on Wizard (Windows). Our project name will be "Converter" and the add-on name will be "Simple Spreadsheet Converter."

When running **`CreateNewAddOn`** or the Tecplot Add-on Wizard answer the questions as follows:

• Project name (base name)	Converter
• Add-on name:	Simple Spreadsheet Converter
• Company Name:	[Your company name]
• Type of add-on:	Data Converter
• Language:	C
• Use TGB to create a platform-independent GUI?:	No
• Add a menu callback to the Tecplot?:	No

After running **`CreateNewAddOn`** or Tecplot Add-on Wizard you should have the following files:
`engine.c`

```
ENGINE.h
main.c
ADDGLBL.h
```

There will be other files specific to your platform, however, we will only be dealing with those above. Verify that the add-on will compile and that it can be loaded into Tecplot. If any problems are encountered, refer to Chapter 2 *Creating Add-ons under Windows* or Chapter 3 *Creating Add-ons under UNIX*.

The file **`ADDGLBL.h`** contains information specific to the add-on, such as its name, version number, and date. The files **`ENGINE.h`** and **`engine.c`** contain the main converter function. **`engine.c`** currently has a short message saying that the converter is under construction. Throughout this tutorial, code will be added to **`engine.c`** so when Tecplot calls the **`ConverterCallback()`** function it will perform of loading the file. The file **`main.c`** contains a function called **`InitTecAddOn()`**. This registers the add-on with Tecplot. Note that within this function there are other function calls which tell Tecplot the name of the add-on and state that it is a converter. The **`InitTecAddOn()`** function is called by Tecplot exactly when the add-on is first loaded, and is not called again.

MODIFYING THE CONVERTERCALLBACK() FUNCTION

When *Converter* is loaded by Tecplot, an option called Simple Spreadsheet Converter will appear in the Import menu of Tecplot’s File menu. When *Converter* is launched, Tecplot will ask for a file to convert. This is the file name that is passed to the **`ConverterCallback()`** function. Tecplot will also create a unique temporary file name and pass that to **`ConverterCallback()`** as well.

In **`ConverterCallback()`** we are required to:

- Open the file **`DataFName`**.
- Convert the data and create a Tecplot binary data file.
- Close the file **`DataFName`**.
- Inform Tecplot if there were any errors.

Note how the **`ConverterCallback()`** function satisfies these requirements:

```
Boolean_t STDCALL ConverterCallback(char  *DataFName,
                                    char  *TempBinFName,
                                    char **MessageString)
{

  Boolean_t IsOk = TRUE;
  FILE *f;

  TecUtilLockStart(AddOnID);

  /* If there is no error, remember to free MessageString. */
  *MessageString = TecUtilStringAlloc(1000,”MessageString for CNVSS”);

  /* Try to open the file. */
  f = fopen(DataFName,”rb”);

  /* Make sure the file was opened. */
  if (!f)
    {
      strcpy(*MessageString,”Cannot open input file.”);
      IsOk = FALSE;
    }
  /* Do the conversion. */
  if (IsOk)
    IsOk = DoConversion(f,TempBinFName,MessageString);

  /* Close the file. */
  fclose(f);

  /* If there was no errors, deallocate MessageString. */
  if (IsOk)
    TecUtilStringDealloc(MessageString);

  TecUtilLockFinish(AddOnID);
  return IsOk;
}
```

This function does the following:

- Creates an error message.* **MessageString** is allocated here because the **DoConversion()** function (which will be explained later) may alter the error message that is reported.
- Attempts to open the file. If the file cannot be opened, it sets **IsOk** to **FALSE**, and resets the ***MessageString** to reflect the fact that the file could not be opened.
- If the file was opened, it converts it. The task of conversion is handed off to the **DoConversion()** function.
- Some clean up is performed, such as closing the file, de-allocating ***MessageString** if there were no errors, and returning **IsOk**. If **IsOk** is **FALSE** at the end of the function, there was an error. Tecplot will use the string in ***MessageString** to display an error message.

WRITING THE DOCONVERSION() FUNCTION

Now that the file is open, we want to perform the conversion. In **ConverterCallback()** the job of performing the conversion is passed to the **DoConversion()** function. **DoConversion()** is responsible for parsing the file to be converted and sending specific information to the **TecUtil** functions which take care of the conversion. A discussion of the **TecUtil** functions is available in the *ADK Reference Manual*. In writing the **DoConversion()** function we are going to make some assumptions about the format of the incoming file: that the variables are at the top of the file, contained in quotes, and separated by commas or spaces; that the data follows the variables and is separated by commas or spaces.

An example of such a file would be:

```
“Var 1” “Var 2” “Var 3”
1.23, 4.4, 3.24
2.45, 3.56, 5.2
3.2, 2.15, 7.56
```

The basic form of a conversion function is:

- Get variable names from the file into a comma-separated string.
- Call **TecUtilTecIni()** to initialize the temporary file.
- Call **TecUtilTecZne()** to add a zone.
- Get data points into an array.
- Call **TecUtilTecDat()** to add the data points to the temporary file.

- Call **`TecUtilTecEnd()`** to close the temporary file.

There are other things that our conversion function will do, however, the steps listed above are the minimum required. There are two functions used for parsing the income file. These are **`GetVars()`** and **`get_token()`**. **`GetVars()`** takes two parameters: a **`FILE*`** and a **`StringList_pa`**. Be sure that you understand the **`StringList_pa`** data type before continuing. For a discussion of **`StringList_pa`** see the *ADK User's Manual*. **`GetVars()`** will parse the text file for the variable names and place them in the string list. **`get_token()`** takes a **`FILE*`** and will parse a text file for items which are separated by commas or spaces. There is no checking to make sure that the item is a valid number. **`get_token()`** will update a global variable called **`_token`**, which is used in **`DoConversion()`**:

```
static Boolean_t DoConversion(FILE  *f,
                              char  *TempFName,
                              char **MessageString)
{
  Boolean_t IsOk = TRUE;
  StringList_pa VarList = TecUtilStringListAlloc(); /* Variable list. */
  int i;
  int NumValues;
  int NumVars;
  int IMax;

  /* First, we need to read all of the variables. */
  GetVars(f,VarList);

  /* Make sure there is at least one variable. */
  if (IsOk && TecUtilStringListGetCount(VarList) < 1)
    {
      strcpy(*MessageString,”No variables defined.”);
      IsOk = FALSE;
    }

  if (IsOk)
    {
      /* Debug and VIsDouble are flags used by TecUtilTecIni(). */
      int Debug = 0;
      int VIsDouble = 1;

      /* Set JMax and KMax to 1 because we are creating an. */
      /* I-ordered data set. */

      int JMax=1,KMax=1;
      char VarNames[5000];
      char *s;

      NumValues = 0;

      /* VarList was filled by the function GetVars. */
      NumVars = TecUtilStringListGetCount(VarList);

      /* Count the number of data points. */
      while (get_token(f))
        {
          NumValues++;
        }

      /*
```

```
 * Get_token() changed where the file pointer is pointing, so
 * we must rewind to the start of the data.
 */
fsetpos(f,&_DataStartPos);

/* Compute the number of data points. */
IMax = NumValues/NumVars;

/* FillVarNames with the variable names in VarList. */
strcpy(VarNames,””);
for (i=1; i<=NumVars && IsOk; i++)
  {
    s = TecUtilStringListGetString(VarList,i);
    strcat(VarNames,s);
    if (i<NumVars)
      strcat(VarNames,”,”);
    TecUtilStringDealloc(&s);
  }

/*
 * Use the TecUtilTecIni() function to initialize the TempFName
 * file and fill it with the data set title and the variable name.
 */
if (TecUtilTecIni(“ConvertedDataset”, VarNames,
                  TempFName,”.”,&Debug,&VIsDouble) != 0)
  {
    strcpy(*MessageString,”Could not create data set.”);
    IsOk = FALSE;
  }

/*
 * Use TecUtilTecZne to add the first zone.
 * In this case, it is the only zone.
 */
if (IsOk && TecUtilTecZne(“Zone 1”,
                          &IMax,&JMax,&KMax,
                          “POINT”,NULL) != 0)
  {
    strcpy(*MessageString,”Could not add zone.”);
    IsOk = FALSE;
  }

/* Now add the data. */
if (IsOk)
  {
    LgIndex_t PointIndex = 1;
    int Skip = 0;

    /* Allocate space to temporarily store the values. */
    double *LineValues = (double*) calloc(NumValues,sizeof(double));

    /* Get the values into the array LineValues. */
    for (i=0; i<NumValues; i++)
      {
        get_token(f);
        LineValues[i] = atof(_token);
      }
```

```
          /*
           * Use the function TecUtilTecDat() to fill the
           * temporary file with the values stored in the LineValues.
           */
          if (TecUtilTecDat(&NumValues,(void*)LineValues,&VIsDouble) != 0)
            {
              strcpy(*MessageString,"Error loading data.");
              IsOk = FALSE;
            }

          /* Free LineValues now that we are done using it. */
          free(LineValues);
        }
    }

  /* Calling TecUtilTecEnd() closes the temporary file. */
  if (TecUtilTecEnd() != 0)
    {
      IsOk = FALSE;
      strcpy(*MessageString,"Error closing temporary file, "
                            "could not create data set.");
    }

  TecUtilStringListDealloc(&VarList);
  return IsOk;
}
```

PARSING THE CODE

A discussion of the parsing of the incoming file is not in the scope of this tutorial. However the parsing code has been included for completness in the sections below.

THE GET_TOKEN() FUNCTION

The get_token() parses the file fetching basic tokens. Here is the function from engine.c:

```
/**
 */
#define MAX_TOKEN_LEN 5000
static char _token[MAX_TOKEN_LEN]; /* Global buffer for tokens. */

/**
 * Get the next token.
 *
 * @param f
 *     Open file handle. The file must be open for binary reading.
 * @return
 *     TRUE if more a token was fetched, FALSE otherwise.
 */
static Boolean_t get_token(FILE *f)
{
  int index = 0;
  char c;
  Boolean_t StopRightQuote;

  /* Skip white space. */
  while (fread(&c,sizeof(char),1,f) == 1 &&
         (c == ‘ ‘ || c == ‘,’ || c == ‘\t’ || c == ‘\n’ || c == ‘\r’))
    {
      /* Keep going. */
    }

  if (!feof(f))
    {
      /* Now we’re sitting on a non-white space character. */
      StopRightQuote = (c == ‘”’);
      if (StopRightQuote)
        {
          _token[index++] = c;
          fread(&c,sizeof(char),1,f);
        }

      do
        {
          if (index == MAX_TOKEN_LEN-1)
            break; /* Lines shouldn’t be longer than 5,000 characters. */

          if (feof(f))
            break;

          if (StopRightQuote)
            {
              if (c == ‘”’)
                {
                  _token[index++] = c;
                  break;
                }
            }
          else
            {
              /* Note that a space or comma may terminate the token. */
              if (c == ‘ ‘ || c == ‘,’ || c == ‘\t’ || c == ‘\n’ || c == ‘\r’)
                break;
            }

          _token[index++] = c;
          fread(&c,sizeof(char),1,f);
        } while(1);
    }

  _token[index] = ‘\0’;

  return (strlen(_token)) > 0;
}
```

THE GETVARS() FUNCTION

This function reads a line of comma- or space-separated variables from the top of the file to be imported. The variables may optionally be enclosed in double quotes.

```
/**
 */
static fpos_t _DataStartPos;

/**
 * Reads a line of comman or space separated variables from the
 * top of the file to be imported. The variables may optionally
 * be enclosed in double quotes.
 *
 * @param f
 *     Open file handle. The file must be open for binary reading.
 * @return
 *     TRUE if more a token was fetched, FALSE otherwise.
 */
static void GetVars(FILE          *f,
                    StringList_pa  sl)
{
  char c;
  char buffer[5000];
  char *Line = buffer;
  char Var[100];
  int  Index = 0;
  char Delimiter = ‘ ‘;

  /* Read up to the first new line. */
  do
    {
      if (fread(&c,sizeof(char),1,f) < 1)
        break;

      if (c != ‘\r’ && c != ‘\n’ && c != ‘\0’)
        buffer[Index++] = c;
      else
        break;
    } while (1);

  buffer[Index] = ‘\0’;

  /* Now get the variable names. */
  while (*Line)
    {
      Index = 0;
      if (*Line == ‘”’)
        {
          /* Skip to next double quote. */
          Line++;
          while (*Line && *Line != ‘”’)
            Var[Index++] = *Line++;
        }
      else
        {
          /* Read to the next delimiter. */
          while (*Line && *Line != Delimiter)
            Var[Index++] = *Line++;
        }

      Var[Index] = ‘\0’;
      TecUtilStringListAppendString(sl,Var);

      /* Skip to the next non-delimiter char. */
      while (*Line && *Line != Delimiter)
        Line++;

      fgetpos(f,&_DataStartPos);

      /* Skip to next non-delimiter char. */
      while (*Line && (*Line == Delimiter || *Line == ‘ ‘))
        Line++;
    }
}
```

Converter is now complete. Recompile and load it into Tecplot.

CHAPTER 9 Creating a Data Loader

LOADERS VERSUS CONVERTERS

Data can be imported into Tecplot using *loader* or *converter* add-ons. A *loader* must display a dialog for the user to enter the parameters needed to load the data; file name, skip values, and so forth. A *converter* is used when simple proprietary data files need to be read into Tecplot and it is not necessary to use complex options to decide which portions of the data should be loaded.

How do add-on loaders work in Tecplot?

A data loader is a special type of add-on which can load data into Tecplot in many customized ways. Data can come from data files, but this is not a requirement. Tecplot provides loaders for several popular file formats, including PLOT3D and Gridgen. In Tecplot, all data loaders appear under the Import option of the File menu. Data loaders usually have custom dialogs for collecting loading parameters.

How does an add-on identify itself as a data loader?

An add-on informs Tecplot that it is a data loader by:

- Registering as a data loader by calling the **`TecUtilImportAddLoader()`** function. This is called from the **`InitTecAddOn`** function in **`main.c`**.
- Exporting a callback function called by Tecplot when you select the Import option from the File menu. (The interface callback.) This usually displays a dialog to collect loading parameters. After collecting the parameters the add-on will call the loader function to load the data.
- Exporting a callback function which is called by Tecplot to load the data. (The loader callback.)

After its been registered, the *loader* add-on waits for:

- It to be selected from the Import option.
- Tecplot to process the **`$!READDATASET`** macro command.

When selected, Tecplot calls the registered interface callback. If processing the **`$!READDATASET`** command, Tecplot calls the loader callback. (In this case the add-on will not display a dialog.)

INTRODUCTION TO THE LOADTXT ADD-ON

In this chapter you will learn methods for structuring your C add-on source code to improve readability and maintenance. After completion you will have the skills needed to write a data loader for your own file formats.

LoadTxt will do the following:

- Open a text file, such as “**`mydata.txt`**.”

- Read the first line of text file for variables. Each variable will be separated by one or more spaces or tabs, as below:

```
Time       Pressure    Temperature
0          34.5        32.0
1          33.4        31.4
2          33.0        31.0
3          31.0        29.4
...        ...         ...
```

- Read the subsequent lines. Each will be a list of values for the *n*th data point, where *n* is the number of additional lines. Values must be separated by one or more spaces or tabs.
- Create a data set in Tecplot with the data from the file.

Since *LoadTxt* is a data loader, we will create a dialog where the user may enter the skip value and the file name. *LoadTxt* will collect this information and use it when reading data into Tecplot. The skip value *n* will be used to read every *n*th data point. We will also implement a macro interface for the **`$!READDATASET`** command.

CREATING LOADTXT

LoadTxt, the Tecplot add-on you will build, is a basic data loader. It will appear under the Import option of Tecplot's File menu as Delimited Text Loader. All of the examples of the source code shown in this manual are included in the Tecplot distribution and are found in the **`adk/samples`** sub-directory below the Tecplot home directory. *LoadTxt* uses source code files created by the **`CreateNewAddOn`** script (UNIX), or Tecplot Add-on Wizard (Windows). Our project and add-on names will be *LoadTxt*.

When running **`CreateNewAddOn`** or Tecplot Add-on Wizard use the following answers:

Project name (base name)	LoadTxt
Add-on name:	LoadTxt
Company Name:	[Your company name]
Type of add-on:	Data Loader
Language:	C
Use TGB to create a platform-independent GUI?:	No
Add a menu callback to the Tecplot?:	No
Data Loader Override:	No

After running the **`CreateNewAddOn`** script or Tecplot Add-on Wizard you should have the following files:

```
engine.c   guibld.c guicb.cguidefs.c
main.c     ADDGLBL.h GUIDEFS.hengine.hgui.lay
```

You will also have other files specific to your platform, but only those above will be modified. Their purpose will be explained as we proceed. At this point, verify that you can compile your project and load it into Tecplot. If not, please see Chapter 2 "Creating Add-ons under Windows," or Chapter 3 "Creating Add-ons under UNIX."

REGISTERING CALLBACKS

Two function prototypes are generated for you in **`ENGINE.h`**:

```
extern Boolean_t STDCALL LoaderCallback(StringList_pa params);
extern void STDCALL LoaderSelectedCallback(void);
```

LoaderCallback() will be called when the **$!READDATASET** macro command is processed. The variable **Instructions** is a string list which will contain the loading instructions. **LoaderSelectedCallback()** will be called only when the Import option is selected from the File menu. Its job will be to display a dialog and collect loading parameters. If you’re not familiar with string lists, refer to the *ADK User's Manual* before proceeding. In order to understand the loader callback function, it is important to understand what string lists are and how they work.

The $!READATASET Interface

Now is a good time to decide what commands can be passed to the loader with the **$!READDATASET** macro command. The text loader will use Tecplot standard syntax so that it integrates better with Tecplot’s ability to make paths relative to a layout file if the **$!READDATASET** command is part of a layout.

LoadTxt instructions will have three name/value pair commands:

Name	Example Value	Required	Default Value
STANDARDSYNTAX	“1.0”	Yes	N/A
FILENAME_TOLOAD	“MyFile.txt”	Yes	N/A
SKIP	“3”	No	“1”

With the above command set we can use the following command in a Tecplot macro:

```
$!READDATSET '”STANDARDSYNTAX” “1.0” "FILENAME_TOLOAD" "MyFile.txt" "SKIP" "3"'
DATASETREADER = "LoadTxt"
```

This tells the loader to read **MyFile.txt** with a skip value of 3. Note the connection between the string list passed to the **LoaderCallback()** function and the parameters we use in the macro command: each quoted string in the macro command is passed as a string in the string list. Tecplot calls the **LoaderCallback()** function with string list **Instructions**. The string list **Instructions** contains six strings: “**STANDARDSYNTAX”, “1.0”, “FILENAME_TOLOAD”,** “**MyFile.txt**”, “**SKIP**” and “**3**”. The main task of the callback function will be to examine each string in the list and determine what the command is. Once the commands are determined, the loader function will call the **DoLoadDelimitedText()** function to load the file. We will define this by adding the following line to **ADDGLBL.h**:

```
Boolean_t DoLoadDelimitedText(const char *FileName, int Skip);
```

Two functions are used because we need a way to load a file from the dialog interface. The OK button callback in the dialog interface, displayed when the Import option is selected from the File menu, will call **DoLoadDelimitedText()** along with the **FileName** and **Skip** parameters from the dialog. Alternatively, the OK button callback could call **LoaderCallback** function directly. In that case you would have to create a StringList_pa object to pass to the callback function. Before the dialog is shown check to see if a data set currently exists in Tecplot. If so, then ask the user if they want to overwrite it. If the answer is “No,” skip the process. At this point we can also stub out the two loader functions and register our loader callbacks.

The **LoaderSelectedCallback()** is in **engine.c** as looks follows:

```
void STDCALL LoaderSelectedCallback(void)
{
  Boolean_t OkToLoad = TRUE;
  TecUtilLockStart(AddOnID);

  if (TecUtilDataSetIsAvailable())
```

```
    OkToLoad = TecUtilDialogMessageBox(“The current data set will “
                                       “be replaced. Continue?”,
                                       MessageBox_YesNo);

  if (OkToLoad)
    {
      BuildDialog1(MAINDIALOGID);
      TecGUIDialogLaunch(Dialog1Manager);
    }

  TecUtilLockFinish(AddOnID);
}
```

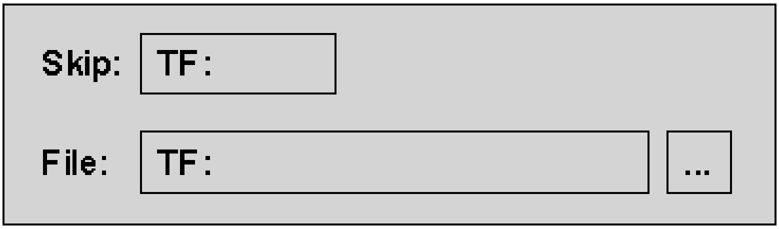
Creating the Dialog

Now create the dialog which will be displayed when users select the Import option from the File menu.

The dialog will be modal and have two fields, one each for the file name and skip value. We will also add a Browse button next to the file name field. Since we are using Tecplot GUI Builder (TGB), the dialog template is the Tecplot layout file **`gui.lay`**.

To do this, perform the steps below.

1. Load **`gui.lay`** into Tecplot, select Tecplot GUI Builder from the Tools menu, then modify the layout to look as follows:

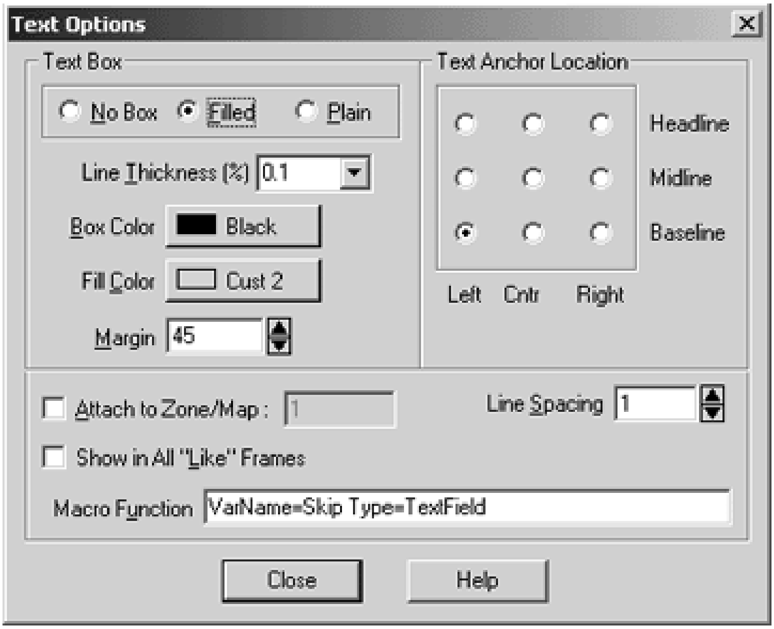


We have added two labels, two text fields, and a button.

There will be two variables associated with the text fields: **`Skip`** and **`FileName`**. So that TGB will create meaningful names for these text fields, we will change the properties of controls.

N**ote:** Although the text fields and buttons are referred to as controls, they are represented by Tecplot text objects, since they exist in a Tecplot layout file.

2. Double-click on the Skip text field and select Options. In the Macro Function field, type Skip. This will be the base name of the skip variable. Do the same for the FileName field.



Text Options

Text Box

○ No Box ● Filled ○ Plain

Line Thickness (%) 0.1

Box Color Black

Fill Color Cust 2

Margin 45

Text Anchor Location

	Left	Cntr	Right
Headline	○	○	○
Midline	○	○	○
Baseline	●	○	○

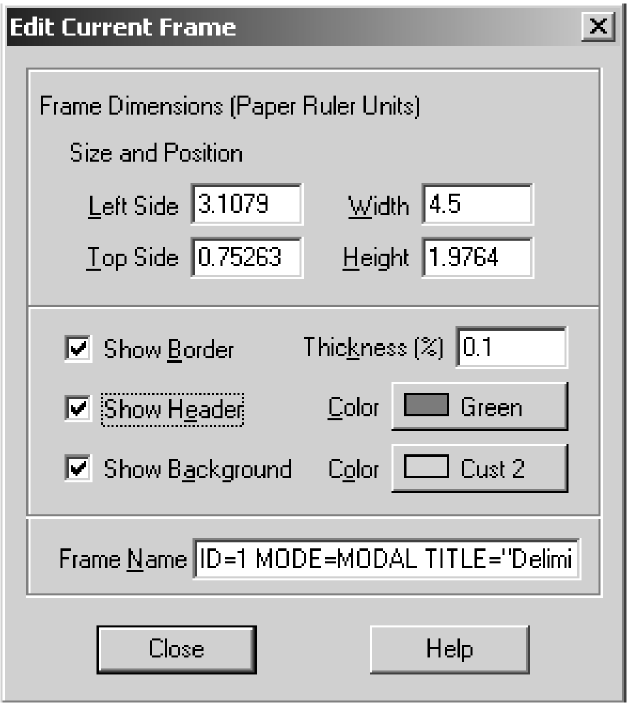
□ Attach to Zone/Map : 1

Line Spacing 1

□ Show in All "Like" Frames

Macro Function VarName=Skip Type=TextField

3. Since the "..." (Browse) button has a callback, change the name of that in TGB to something more meaningful. Double-click on "..." and select Options. In the Macro Function Field, type Browse. TGB will use this name when creating the callback function.



Edit Current Frame

Frame Dimensions (Paper Ruler Units)

Size and Position

Left Side 3.1079 Width 4.5

Top Side 0.75263 Height 1.9764

☑ Show Border Thickness (%) 0.1

☑ Show Header Color Green

☑ Show Background Color Cust 2

Frame Name ID=1 MODE=MODAL TITLE="Delimi

4. Now give the dialog a title. In TGB, the title of the dialog is specified in the Edit Current Frame dialog. Double click on the dialog frame and set Frame Name to:

```
ID=1 MODE=MODAL TITLE=”Delimited Text Loader”
```

5. You can now build the source for this layout. From the TGB dialog click Go Build.

6. Rename the file **`guicb.tmp`** to be **`guicb.c`**.

SETTING UP STATE VARIABLES/INITIALIZING DIALOG FIELDS

The loader function requires two parameters. For convenience, the loader will remember the previous skip value and file name during the same Tecplot session; we will not attempt to remember the values between Tecplot sessions. In the next step we will set up global variables which will remember those values from one invocation of the loader to the next during the same Tecplot session.

Normally, global variables should be kept to a minimum. Since having global variables cannot be avoided in this case, we will create a structure which will contain all of the global variables. Then we will declare exactly one of these structures and call it **`GS`**. Thus, any place in the code where a global variable is referenced will be immediately obvious because the reference will always be scoped with **`GS.GlobalVariable.`**

1. Note the following lines in **`ADDGLBL.h`**:

```
#define MAX_FILENAME 5000
typedef struct
{
  char FileName[MAX_FILENAME];
  int  Skip;
} LoadTxtGlobalState_s;
```

2. the lines in **`main.c`**:

```
LoadTxtGlobalState_s GS;
```

3. and the line in **`guicb.c`**:

```
extern LoadTxtGlobalState_s GS;
```

Next initialize the global state. The logical place to do this is in the **`InitTecAddOn()`** function, which is called by Tecplot only once when the add-on is first loaded. Note the following lines in **`InitTecAddOn()`** found in **`main.c`**:

```
/*
 * Initialize the file name to be empty and the default skip
 * parameter to hav a value of 1.
 */
GS.FileName[0]  = ‘\0’;
GS.Skip         = 1;
```

The values in the dialog are synchronized with the global state in the **`Dialog1Init_CB()`** function found in **`guicb.c`**:

```
static void Dialog1Init_CB(void)
{
  char StrSkip[32];
  TecUtilLockStart(AddOnID);

  sprintf(StrSkip,”%d”,GS.Skip);

  TecGUITextFieldSetString(Skip_TF_D1,StrSkip);
  TecGUITextFieldSetString(FileName_TF_D1,GS.FileName);
}
```

IMPLEMENTING DIALOG CALLBACKS

In this step we will modify the callbacks in the dialog so that it collects the file name and skip parameters. We will also verify that those parameters are valid. Finally we will call our unfinished **`DoLoadDelimitedText()`** function from the dialog’s OK callback.

The FileName Text Field Callback

When a text field loses the focus, a callback is received if the text has changed since the text field received the focus. We could check that the file name is valid at that time, but instead we will check the validity of the file name in the OK button callback. Thus the function **`int FileName_TF_D1_CB()`** will remain empty.

The Skip Text Field Callback

When the skip callback is received, we need to check that the skip value entered in the dialog is valid. The skip value will be valid if it is an integer greater than or equal to one.

All text field callbacks in TGB return an **`int`** value. If data in the text field is valid, the callback should return one, otherwise it should return zero. This allows TGB to replace the text with the last known valid entry if the entered text is invalid. Examine the following lines in **`guicb.c`**:

```
static int  Skip_TF_D1_CB(const char *S)
{
  int IsOk = 1;
  int Value;

  TecUtilLockStart(AddOnID);

  /*
   * We could be more elaborate here and check for non-numeric
   * digits, but for now just check that the string converts to
   * an integer >= 1
   */
  Value = atoi(S);
  if (Value < 1)
    {
      TecUtilDialogErrMsg(“Skip parameter must be greater than “
                          “or equal to 1.”);
      IsOk = 0;
    }
  if (IsOk)
    GS.Skip = Value; /* Remember for next time */

  TecUtilLockFinish(AddOnID);

  return IsOk;
}
```

The Browse Button Callback

The browse button will display a file dialog and allow the user to select a file. When you click OK in the browse dialog, the **`FileName`** text field in the main dialog should be filled in with the selected file.

Examine the **`Browse_BTN_D1_CB()`** callback function in **`guicb.c`**:

```
static void Browse_BTN_D1_CB(void)
{
  char *SelectedFileName  = NULL;
  char *Type              = "Delimited Text";
  char *Filter            = "*.txt";

  TecUtilLockStart(AddOnID);

  if (TecUtilDialogGetFileName(SelectFileOption_ReadSingleFile,
                               &SelectedFileName,Type,GS.FileName,Filter))
    {
      strcpy(GS.FileName,SelectedFileName);
      TecUtilStringDealloc(&SelectedFileName);
      TecGUITextFieldSetString(FileName_TF_D1,GS.FileName);
    }

  TecUtilLockFinish(AddOnID);
}
```

The **`TecUtilDialogGetFileName()`** function will display a file dialog and allow the user to select a file. If you click OK the function will return **`TRUE`**, otherwise you have clicked Cancel. We pass **`TecUtilDialogGetFileName()`** an address of a **`char *`**, which receives the file name. Only if the function returns **`TRUE`** are we required to release the string using **`TecUtilStringDealloc()`**. Before releasing the string it is copied into the global state structure.

The OK Button Callback

Now we must:

- Call the loader function, **`DoLoadDelimitedText()`** with the file name and skip parameters collected from the dialog.
- If the loader function returns **`TRUE`**, save the file name in the global state.

In **`guicb.c`**, the modified **`Dialog1OkButtonCallback()`** function looks as follows:

```
static void Dialog1OkButton_CB(void)
```

```
{
  char *FileName;

  FileName = TecGUITextFieldGetString(FileName_TF_D1);

  /* check that the filename is valid */
  if ((FileName != NULL && strlen(FileName) > 0) &&
      DoLoadDelimitedText(FileName,GS.Skip))
    {
      /* Save the filename for next time */
      strcpy(GS.FileName,FileName);
      TecGUIDialogDrop(Dialog1Manager);
      TecUtilLockFinish(AddOnID);
    }

  if (FileName != NULL)
    {
      TecUtilLockStart(AddOnID);
      TecUtilStringDealloc(&FileName);
      TecUtilLockFinish(AddOnID);
    }
}
```

If there are any errors loading the file, then **DoLoadDelimitedText()** will return **FALSE**, and we will not drop the dialog. **DoLoadDelimitedText()** will display any error message, so we will not do that in the callback. We have to release the string returned by **TecGUITextFieldGetString()**.

PREPARING TO WRITE THE LOADER FUNCTION

In this step we will write the **DoLoadDelimitedText()** function. This will read the text file and load it into Tecplot, returning **TRUE** if successful.

File Format

If we were writing a loader for a known file format, such as an Excel spreadsheet file, we could study the specification for that format and decide how best to code the loader function. In this case, however, there is no previously defined specification, so we will have to write our own. The more detailed we are in how the file can be formatted, the easier it will be to write the function. It is useful to impose rules on the specification which make it easier to make assumptions about the format of the file. These assumptions can then be used to our advantage in the source code.

The first general rule of our text file format will be that blank lines are ignored, but any non-blank line must be a valid part of the file.

Note the function, **IsBlankLine()**, which takes a character string representing a line and returns **TRUE** if the line is blank. We will define a blank line as zero or more tabs or spaces followed by a new line (**\n**) and/or carriage return (**\r**). The following lines are in **engine.c**:

```
static Boolean_t IsBlankLine(const char *Line)
{
  Boolean_t Result = TRUE;
  int       Index = 0;
  REQUIRE(VALID_REF(Line));

  /*
   * Search through the string for any character that is not:
   * “ \t\r\n”.
   */

  while (Result && Line[Index] != ‘\0’)
    {
      /* These are the only characters in a blank line */
      Result = (Line[Index] == ‘ ‘  ||
                Line[Index] == ‘\t’ ||
                Line[Index] == ‘\n’ ||
                Line[Index] == ‘\r’);
      Index++;
    }

  return Result;
}
```

Next we will require that all variables appear on the first non-blank line in the file, separated by spaces or tabs. This rule implies that variable names cannot contain spaces.

The **`GetVariableNames()`** function takes a string list object and a character string containing the line, and fills in the string list with all of the variable names. We will make use of the C standard library **`strtok()`** function:

```
static void GetVariableNames(StringList_pa  VarList,
                             char          *Line)
{
  /* NOTE: We assume Tecplot is locked */
  char *Token;

  REQUIRE(VALID_REF(Line));

  /* Be sure that the string list is initially empty */
  TecUtilStringListClear(VarList);

  /* Note that strtok() will write into Line */
  Token = strtok(Line,” \t\r\n”);

  while (Token)
    {
      /* Token now points to a variable name, so add it to the list */
      TecUtilStringListAppendString(VarList,Token);
      Token = strtok(NULL,” \t\r\n”); /* Get the next one */
    }
}
```

You can see that we are looking for groups of text separated by either tabs, spaces, or the end of the line. Each group of text (**`Token`**) is a variable name, and is added to the string list. We do not have to keep track of the total number of variable names because the string list object will do this for us. If needed, we can call **`TecUtilStringListGetCount()`** and find out the total number of strings in the string list.

How should the input data be interpreted by Tecplot?

With the tools above we are ready to write the loader function. It is useful at this point to think about how Tecplot should interpret the imported data. Since the input files have a row of variables followed by one or more rows of values, we will import the text file as an I-ordered data set. Line 1 of the file is the list of variable names. The following lines will contain the values of the variables at each data point. For example, suppose the input file has three lines as follows:

```
Var1            Var2            Var3
1.0             3.4             1.2
5.4             2.3             4.5
```

In this case our loader reads the first line and finds three variables. Line 2 of the file contains the values of the variables at data point 1, and so forth.

Before creating a data set in Tecplot, we need to know the **I**-dimension, which in this case will be a function of the skip value and the number of non-blank lines in the file. Thus it will be necessary to make two passes through the file.

To buffer or not to buffer the data?

There are two ways to process the data.

Data Processing Method 1

During the first pass, allocate memory for the values and read them as we go along. Disk input/output is minimized, and the second pass is much faster, since the values are already in memory at that point. We can also do some error checking after the first pass before we create the data set in Tecplot.

The disadvantage of this method is that for a short time during the second pass there are two complete copies of the data set in memory at once. This is not a problem unless you anticipate that your data files will be extremely large (in the multi-megabyte range).

Data Processing Method 2

During the first pass count the number of lines.   Combined with the skip value, this will determine the **I**-dimension of the data set. During the second pass rewind the file and read the values again, starting at the line after the variables. No memory is wasted, even temporarily. This method is easier to code, since it is not necessary to decide how the values will be stored in memory between the passes.

For our purposes there are no significant disadvantages to the second method so we will use it to read the input files.

EXPORT FUNCTIONS

`Pass1` uses the **`IsBlankLine()`** and **`GetVariableNames()`** functions.

Note the following lines in **`engine.c`**:

```
static Boolean_t Pass1(int           *LineCount,
                       StringList_pa  VarNames,
                       FILE          *F)
{
  Boolean_t IsOk = TRUE;
  char Line[MAX_LINE_SIZE];

  REQUIRE(VALID_REF(LineCount));
  REQUIRE(VALID_REF(VarNames));
  REQUIRE(VALID_REF(F));
  while (fgets(Line,MAX_LINE_SIZE,F) != NULL)
    {
      if (!IsBlankLine(Line))
        {
          /* This must be the line with the variable names */
          GetVariableNames(VarNames,Line);
          break;
        }
    }
  /* Must be at least one variable */
  if (TecUtilStringListGetCount(VarNames) == 0)
    {
      TecUtilDialogErrMsg(“No variables specified”);
      IsOk = FALSE;
    }

  if (IsOk)
    {
      /*
       * Now count all the lines.
       */

      *LineCount = 0;

      while (fgets(Line,MAX_LINE_SIZE,F) != NULL)
        {
          if (!IsBlankLine(Line))
            {
              *LineCount += 1;
            }
        }

      if (*LineCount == 0)
        {
          TecUtilDialogErrMsg(“No data specified”);
          IsOk = FALSE;
        }
    }

  return IsOk;
}
```

This function's purpose is to count the number of non-blank lines, excluding the variable line. This count is passed back to the calling function through the **`LineCount`** parameter.

The only error condition we check for during pass 1 is that there be at least one non-blank data line.

For **`Pass2`** note the following lines in **`engine.c`**:

```
static Boolean_t Pass2(int            Skip,
                       int            LineCount,
                       StringList_pa  VarNames,
                       FILE          *F)
{
  Boolean_t IsOk = TRUE;
  char Line[MAX_LINE_SIZE];
  EntIndex_t i;
  EntIndex_t VarCount;
  EntIndex_t PointIndex;
  FieldDataType_e fd_types[MAX_VARIABLES];
  int CurrentLine;
  int IMax;

  REQUIRE(Skip >= 1);
  REQUIRE(LineCount > 0);
  REQUIRE(VALID_REF(VarNames) &&
          TecUtilStringListGetCount(VarNames) > 0);
  REQUIRE(F != NULL);

  /*
   * At this point we know that there is a least one variable and at lease
   * one line of data. So it's safe to create the dataset.
   *
   * Note that once we create the dataset, we are committed to adding the
   * correct amount of data (otherwise we will leave Tecplot in an invalid
   * state), so we must be prepared to default some datapoints to 0 if they
   * cannot be read from the file.
   */
  VarCount = TecUtilStringListGetCount(VarNames);

  /*
   * The data type for each value we read will be double.
   */
  for (i=0;i<VarCount;i++)
    fd_types[i] = FieldDataType_Double;

  rewind(F);

  /*
   * Skip to the first non-blank line.
   * This is the line with the variables.
   */
  while (fgets(Line,MAX_LINE_SIZE,F))
    {
      if(!IsBlankLine(Line))
        break;
    }

  if (LineCount > 1)
    IMax = 2 + (div(LineCount-2,Skip)).quot;
  else
    IMax = 1;

  IsOk = (IsOk &&
          TecUtilDataSetCreate(“Converted Text Dataset”,
                               VarNames, TRUE) &&
          TecUtilDataSetAddZone(“Zone 1”,IMax,1,1,
                                ZoneType_Ordered, fd_types));


  if (IsOk)
    {
      CurrentLine   = 1;
      PointIndex    = 1;

      TecUtilDialogLaunchPercentDone(“Importing...”,TRUE);

      do
        {
          /* Get a line */
          if (fgets(Line,MAX_LINE_SIZE,F))
            {
              /* Is it a blank line */
              if (IsBlankLine(Line))
                continue; /* Skip this line */

              /* Always include the first and last points when skipping */
              if ((CurrentLine-1) % Skip == 0 ||
                  CurrentLine == 1 ||
                  CurrentLine == LineCount)
                {
                  AddDataPoints(PointIndex,Line,VarCount);
                  PointIndex += 1;
                }
            }
          else
            {
              break; /* Done */
            }

          TecUtilDialogCheckPercentDone(MIN(100,
                                            ((int)(100.0*CurrentLine) /
                                             LineCount)));
          CurrentLine += 1;

        } while (TRUE);

      TecUtilDialogDropPercentDone();
    } /* end if */


  return IsOk;
}
```

Adding Field Data

The general steps for adding field data to Tecplot are:

1. Call **`TecUtilDataSetCreate(…).`**
2. Call **`TecUtilDataSetAddZone(…)`** one or more times.
3. Call **`TecUtilDataValueSetByRef(…)`** for each data point in the zone.
4. If the data is finite-element use **`TecUtilDataNodeSetByRef(…).`**

Each variable will be of type double, so they will first create an array of field data types, one for each variable. The value of each data type is set to the constant **`FieldDataType_Double`**. The array of field data types is passed to **`TecUtilDataSetAddZone()`**, and it serves the same function as the **`DT`** parameter in Tecplot ASCII data files. The function then rewinds the file and skips past the line containing the variable names. It will read the field values beginning on the next line.

Skipping

This function will compute the **I**-dimension, since this is required when adding a zone. The **I**-dimension is computed using the skip value.

The Skip Value

By skip value we mean that for a skip value *n*, every *n-1* data points will be skipped. For example, a skip value of 2 would mean that we read data points 1, 3, 5, …, and so forth. For a skip value of 3 we would read data points 1, 4, 7, …, and so forth. A skip value of 1 means that no data points are skipped. Also, the first and last data points are always included irrespective of the skip value.

The **I**-dimension of the data set is computed using the skip value as follows:

```
if (LineCount > 1)
  IMax = 2 + (div(LineCount-2,Skip)).quot;
else
  IMax = 1;
```

Note that **`LineCount`** is the total number of data points. The **I**-dimension is equal to 2 (first and last points) plus the quotient of **`[(LineCount-2)`** divided by the **`Skip`** value]. **`LineCount-2`** is used because we have already included the first and last points in the calculation. The only special case is if there is only one data point. In that case the **I**-dimension is always 1.

For the one-dimensional data set we are creating, this calculation is relatively straightforward. For two- or three-dimensional data sets the calculation would be more complex.

Data Set Creation/Adding Field Data to Tecplot

Now we call **`TecUtilDataSetCreate()`** and **`TecUtilDataSetAddZone()`** to create the data set. Each line of the file is read, then possibly discarded (if it is blank or should be skipped), and imported into Tecplot. To make adding the field data to Tecplot easier, we will use a special function, **`AddDataPoints()`**:

```
static void AddDataPoints(LgIndex_t   PointIndex,
                          const char *Line,
                          EntIndex_t  NumVars)
{
  /* Zone is always 1 */
  FieldData_pa FD;
  EntIndex_t i;
  char buffer[MAX_LINE_SIZE];
  double  Value;
  char *strDataPoint;

  REQUIRE(PointIndex > 0);
  REQUIRE(VALID_REF(Line));
  REQUIRE(NumVars > 0);

  strcpy(buffer,Line);

  /* must have a space between data points */
  strDataPoint = strtok(buffer,” \t\r\n”);

  for (i=1;i<=NumVars;i++)
    {
      FD = TecUtilDataValueGetRef(1,i);
      if (strDataPoint)
        {
          Value = atof(strDataPoint);
          /* get the next one */
          strDataPoint = strtok(NULL,” \t\r\n”);
        }
      else
        {
          /* Default: not enough values on this line */
          Value = 0.0;
        }

      TecUtilDataValueSetByRef(FD,PointIndex,Value);
    }
}
```

Adding field data is straightforward. We utilize an abstract reference to the zone and variable number using **`TecUtilDataValueGetRef()`**, and set the field value using **`TecUtilDataValueSetByRef()`**. Since we have only added one zone, the zone number is always 1. Note that all indices are 1-based.

THE MAIN LOADER FUNCTION

At this point we have written the **`Pass1()`** and **`Pass2()`** functions. Now combine them in the loader function by adding the following lines to **`engine.c`**:

```
Boolean_t DoLoadDelimitedText(const char *FileName,
                              int         Skip)
{
  Boolean_t IsOk    = TRUE;
  FILE      *F      = NULL;
  StringList_pa VarNames;
  StringList_pa LoaderInstructions;
```

```
int       LineCount;

REQUIRE(VALID_REF(FileName) && strlen(FileName) > 0);
REQUIRE(Skip >= 1);

TecUtilLockStart(AddOnID);
VarNames = TecUtilStringListAlloc();

/* Try to open the file */
F = fopen(FileName,”r”);

if (F == NULL)
  {
    TecUtilDialogErrMsg(“Cannot open file for reading”);
    IsOk = FALSE;
  }


IsOk = (IsOk                         &&
        Pass1(&LineCount,VarNames,F) &&
        Pass2(Skip,LineCount,VarNames,F));

if (F)
  fclose(F);

if (IsOk)
  {
    char strSkip[256];

    LoaderInstructions = TecUtilStringListAlloc();
    sprintf(strSkip,”%d”,Skip);

    /*
     * NOTE:
     *
     * The string written to the layout file will look like:
     *
     *  ‘”STANDARDSYNTAX” “1.0” “FILENAME_TOLOAD” “myfile.txt” “SKIP” “3”’
     *
     * This is the recommended way to encode export parameters in
     * the instruction string.
     *
     */
    TecUtilStringListAppendString(LoaderInstructions,STANDARDSYNTAX);
    TecUtilStringListAppendString(LoaderInstructions,”1.0”);
    TecUtilStringListAppendString(LoaderInstructions,FILENAME_TOLOAD);
    TecUtilStringListAppendString(LoaderInstructions,FileName);
    TecUtilStringListAppendString(LoaderInstructions,SKIP);
    TecUtilStringListAppendString(LoaderInstructions,strSkip);

    TecUtilImportSetLoaderInstr(ADDON_NAME,LoaderInstructions);
    TecUtilFrameSetPlotType(PlotType_XYLine);
```

```
        TecUtilRedraw(TRUE);

        TecUtilStringListDealloc(&LoaderInstructions);
      }

    TecUtilStringListDealloc(&VarNames);
    TecUtilLockFinish(AddOnID);

    return IsOk;
  }
```

If the file is loaded successfully we call `TecUtilImportSetLoaderInstructions()`. This function is where we inform Tecplot of the instructions necessary to read the file. If you save a layout, these instructions will be written into the file. When the layout is read back in, Tecplot will call the registered loader function with those instructions.

This brings us to the registered loader function in `engine.c`:

```
  Boolean_t STDCALL LoaderCallback(StringList_pa Instructions)
  {
    Boolean_t IsOk = TRUE;
    LgIndex_t NumParams;
    char      *FileName = NULL;
    int       Skip      = 1; /* default */

    TecUtilLockStart(AddOnID);

    NumParams = TecUtilStringListGetCount(Instructions);
    if (NumParams != 0)
      {
        const char *ParamStrRef = TecUtilStringListGetRawStringPtr(
                                            Instructions, 1);
        if (Str_ustrcmp(ParamStrRef, STANDARDSYNTAX) == 0)
          IsOk = ParseNewSyntax(Instructions, &FileName, &Skip);
        else
          IsOk = ParseOldSyntax(Instructions, &FileName, &Skip);
      }

    if (IsOk)
      {
        if (FileName != NULL && strlen(FileName) > 0)
          DoLoadDelimitedText(FileName,Skip);
        else
          TecUtilDialogErrMsg("No filename specified");

        if (FileName != NULL)
          TecUtilStringDealloc(&FileName);
      }

    TecUtilLockFinish(AddOnID);

    /*
     * Note that you do not need to dealloc the string list 'Instructions' as
    * this will be done by Tecplot or the calling function
    */

  return IsOk;
}
```

It is important to understand the difference between this function, which is the registered loader callback, and the previous function **DoLoadDelimitedText()**, which does the work of loading the data. The registered loader callback is the function that Tecplot calls when processing a **$!READDATASET** macro command. The instruction string passed to it has all of the information needed to load the file. The callback parses the parameters and passes them to **DoLoadDelimitedText()**. The reason for this is that we also want to be able to use **DoLoadDelimitedText()** from the dialog's OK button callback.

Another interesting part of the loader callback is the instruction parsing. The loader callback examines the first name in the instruction list to determine if the loader is using Tecplot's new standard syntax (see the section "Using Standard Instruction Syntax" in the Tecplot ADK User's Manual for a discussion of using the standard instruction syntax). This add-on was written before Tecplot standard instruction syntax existed so it must support both the old syntax and the new. All new add-on's can simply use the standard instruction syntax. In short, using the standard instruction syntax provides an add-on tighter integration with Tecplot's load data options such as the use of relative paths and loading of alternate datasets. For completeness the functions that perform the parsing of the new and old syntax from **engine.c** are included below:

```
/**
 * Add-on's using the standard syntax have better integration with Tecplot.
 *
 * New Syntax:
 *
 *     Name                Value          Required   Default
 *     -----------------------------------------------------
 *     STANDARDSYNTAX      1.0            Yes        N/A
 *     FILENAME_TOLOAD     "myfile.txt"   Yes        N/A
 *     SKIP                3              No         1
 */
static Boolean_t ParseNewSyntax(StringList_pa   NewInstructions,
                                char          **FileName,
                                int            *Skip)
{
  Boolean_t IsOk;
  LgIndex_t NumParams;
  Boolean_t FileNameFound = FALSE;
  Boolean_t SkipFound     = FALSE;

  REQUIRE(VALID_REF(NewInstructions));
  REQUIRE(VALID_REF(FileName) && *FileName == NULL);
  REQUIRE(VALID_REF(Skip) && *Skip == 1);

  NumParams = TecUtilStringListGetCount(NewInstructions);
  IsOk = (NumParams == 4 || NumParams == 6);
  if (IsOk)
    {
      LgIndex_t i;
      /*
```

```
       * The first name value pair is “STANDARDSYNTAX” / “1.0”; start on
       * the second name value pair at position 3.
       */
      for (i=3;i<=NumParams;i+=2)
        {
          const char *ParamStrRef = TecUtilStringListGetRawStringPtr(
                                        NewInstructions,i);
          if (Str_ustrcmp(FILENAME_TOLOAD, ParamStrRef) == 0)
            {
              IsOk = (!FileNameFound);
              if (IsOk)
                {
                  *FileName = TecUtilStringListGetString(
                                  NewInstructions,i+1);
                  FileNameFound = TRUE;
                }
              else
                TecUtilDialogErrMsg(“Cannot specify the FileName_ToLoad “
                                    “text loader option twice.”);
            }
          else if (Str_ustrcmp(SKIP, ParamStrRef) == 0)
            {
              IsOk = (!SkipFound);
              if (IsOk)
                {
                  const char *SkipStrRef =
                      TecUtilStringListGetRawStringPtr(NewInstructions,i+1);
                  *Skip = atoi(SkipStrRef);
                  SkipFound = TRUE;
                }
              else
                TecUtilDialogErrMsg(“Cannot specify the Skip text “
                                    “loader option twice.”);
            }
          else
            {
              IsOk = FALSE;
              TecUtilDialogErrMsg(“Unknown text loader option.”);
            }
        }
    }
  else
    TecUtilDialogErrMsg(“Standard loader syntax expects “
                        “name/value pairs.”);

  return IsOk;
}
/**
 * The old syntax is here for backward compatability. Now we use the
 * Tecplot’s new standard syntax. See ParseNewSyntax().
 *
 * Old Syntax:
```

```
 *
 *     Flag                Value          Required   Default
 *     ------------------------------------------------------
 *     -F                  "myfile.txt"   Yes        N/A
 *     -I                  3              No         1
 */
static Boolean_t ParseOldSyntax(StringList_pa   OldInstructions,
                                char          **FileName,
                                int            *Skip)
{
  LgIndex_t i;
  LgIndex_t NumParams;
  Boolean_t FileNameFound = FALSE;
  Boolean_t SkipFound     = FALSE;

  REQUIRE(VALID_REF(OldInstructions));
  REQUIRE(VALID_REF(FileName) && *FileName == NULL);
  REQUIRE(VALID_REF(Skip) && *Skip == 1);

  NumParams = TecUtilStringListGetCount(OldInstructions);
  for (i=1;i<=NumParams;i++)
    {
      const char *ParamStrRef = TecUtilStringListGetRawStringPtr(
                                    OldInstructions,i);
      if (ParamStrRef[0] == '-'    &&
          strlen(ParamStrRef) >= 2 &&
          i < NumParams)
        {
          /* Found a parameter */
          switch(toupper(ParamStrRef[1]))
            {
              case 'F':
                {
                  /* Found filename */
                  if (!FileNameFound)
                    {
                      *FileName = TecUtilStringListGetString(
                                      OldInstructions,++i);
                      FileNameFound = TRUE;
                    }
                } break; /* 'F' */


              case 'I':
                {
                  if (!SkipFound)
                    {
                      const char *SkipStrRef =
                          TecUtilStringListGetRawStringPtr(
                              OldInstructions,++i);
                      *Skip = atoi(SkipStrRef);
                      SkipFound = TRUE;
```

```
                                }
                            } break; /* 'I' */

                          default:
                            {
                              /* ignore unrecognized option */
                            } break;
                        }
                    }
                }

            return TRUE;
          }
```

LoadTxt is now complete. Recompile and load it into Tecplot.

EXERCISES

1. Modify **`GetVariableNames()`** to accept names delimited by commas instead of spaces. This would allow spaces in variable names, but not commas.

2. Modify **`GetVariableNames()`** to accept names in quotes. This would allow spaces in variable names, but not double quotes.

3. Modify **`Pass1()`** to check for the correct number of variables on each line, or check that each of these variables is numeric (that is, consists only of valid numeric characters). Remember that in addition to digits, valid floating point characters include **`E`** and **`e`**, and (**`.`**).

4. Modify the loader to allow Tecplot-style comment lines in the input file. Any line starting with an octothorp (**`#`**) would be ignored.

5. Modify the loader to accept C-style comments in the input file.

6. Instead of using two loader functions, remove **`DoLoadDelimitedText()`** and modify the callback function **`LoadDelimitedText()`** to load the data. In the OK button dialog callback, create a fake instruction string with the parameters entered in the dialog and call **`LoadDelimitedText()`**.

7. Modify the loader to read all of the data into memory during pass 1, and write the data from memory in pass 2. Combine this with exercise 2 for additional error checking.

8. Allow data values to be separated by commas.

9. In Tecplot ASCII files, any line may contain any number of data values. In our loader, each line of the input file must have exactly one value for each variable. Modify the loader to accept any number of data points on any line of the input file.

10. Modify the loader to remember the last used file name and skip values between Tecplot sessions.

CHAPTER 10 Extending Interactive User Interface Capabilities

INTRODUCTION TO THE SUMPROBE ADD-ON

SumProbe, the add-on you will create in this chapter, is an example of an add-on which can sum up the probed values of a selected variable. It will appear in Tecplot's Tools menu as Sum Probed Values. When selected, a dialog will appear allowing you to specify which variable you wish to sum.

All of the examples of the source code shown in this manual are included in the Tecplot distribution and are found in the **`adk/samples`** sub-directory below the Tecplot home directory.

SumProbe uses source code files created by the **`CreateNewAddOn`** script (UNIX), or Tecplot Add-on Wizard (Windows).

When running **`CreateNewAddOn`** or Tecplot Add-on Wizard answer the questions as follows:

• Project Name	SumProbe
• Add-on name:	Sum Probe
• Company name:	[Your company name]
• Type of add-on:	General Purpose
• Language:	C
• Use TGB to create a platform-independent GUI?	Yes
• Add a menu callback to the Tecplot "Tools" menu?	Yes
• Menu text:	Sum Probed Values
• Menu callback option:	Launch a modeless dialog
• Dialogue Title:	Sum Probe

We will use a **`TecUtil`** function to get the variable name to sum and TGB to create a dialog to display the total number of summed points.

After running **`CreateNewAddOn`** or Tecplot Add-on Wizard you have the following files:

```
guibld.c       guicb.c     guidefs.c    main.c
ADDONGBL.h     GUIDEFS.h   gui.lay
```

You will also have other files specific to your platform, but we will only modify those above. The purpose of each file will be explained in detail as we proceed.

Verify that you can compile your project add-on and load it into Tecplot. If you cannot, refer to Chapter 2 "Creating Add-ons under Windows," or Chapter 3 "Creating Add-ons under UNIX."

THE MENUCALLBACK() FUNCTION

Most add-ons contain a callback function named **MenuCallback()**. This is called by Tecplot each time the add-on is selected from the Tools menu. **MenuCallback()** stores the code that performs all functions of the add-on. This callback function is specified in the **TecUtilMenuAddOption** function, and passes it to Tecplot in **InitTecAddOn()**.

The **TecUtilDialogGetVariables** function has a built-in dialog which allows you to select the variable to be summed. Then the newly-created dialog appears. As points are probed, the summed total is displayed on the dialog.

Before adding the code below, create a label on the dialog which will be set to the total as the plots are probed. (See the *TGB Reference Manual* for more information on adding a label to a TGB dialog.) Set the variable name of this label to VarName=**Totalis00**. Set the text string of the label to read “The total is 0.0”

The new or modified source code is displayed in bulleted lines. If you are working along, add or edit bulleted lines only. (All **TecUtil** functions are defined in the *ADK Reference Manual*.)

Note the **MenuCallback()** function in **main.c**:

```
static void STDCALL MenuCallback(void)
{
  TecUtilLockStart(AddOnID);

  if (TecUtilDataSetIsAvailable())
    {
      if (TecUtilFrameGetPlotType() == PlotType_Cartesian2D)
        {
          TecUtilDialogGetVariables(“Pick Variable to Sum”,
                                    NULL,
                                    NULL,
                                    NULL,
                                    &Variable,
                                    NULL,
                                    NULL);

          BuildDialog1(MAINDIALOGID);
          TecGUIDialogLaunch(Dialog1Manager);

          TecUtilProbeInstallCallback(MyProbeCallback,
                                      “Suming Probed Values”);
        }
      else
        TecUtilDialogErrMsg(“Plot type must be 2D cartesian.”);
    }
  else
    TecUtilDialogErrMsg(“Frame does not contain a dataset “
                        “with which to probe.”);

  TecUtilLockFinish(AddOnID);
}
```

This example is limited to 2-D plots.

THE MYPROBECALLBACK() FUNCTION

The **`TecUtilProbeInstallCallback`**(**`MyProbeCallback, “Summing Probed Values”`**) function calls the function **`MyProbeCallback`** each time a point is probed.

In **`main.c`** note the function **`MyProbeCallback()`** above **`MenuCallback()`**:

```
static void STDCALL MyProbeCallback(Boolean_t IsNearestPoint)
{
  TecUtilLockStart(AddOnID);

  if (IsNearestPoint)
    {
      double ProbeValue = TecUtilProbeFieldGetValue(Variable);
      char Msg[100];

      Total = Total + ProbeValue;
      sprintf(Msg, “The total is: %f”, Total);
      CHECK(strlen(Msg) < sizeof(Msg));

      TecGUILabelSetText(Thetotalis00_LBL_D1, Msg);
    }
  else
    TecUtilDialogErrMsg(“You must hold down the Ctrl key when probing”);

  TecUtilLockFinish(AddOnID);
}
```

Each time a point is probed the callback checks to see if it was probed while holding down Ctrl. If it was, it gets the value of the variable, adds it to the running total, and changes the text displayed on the dialog to reflect this.

SumProbe is complete. Recompile and load it into Tecplot.

EXERCISES

1. Enhance *SumProbe* to allow for interpolated values while probing.

2. Add a Clear button to the dialog to zero out the summed values.

CHAPTER 11 Animating

INTRODUCTION TO THE ANIMIPLANES ADD-ON

AnimIPlanes, the add-on you will create in this chapter, is an example of an add-on which can animate the I-planes of a selected set of zones. It will appear in Tecplot's Tools menu as Animate I Planes. *AnimIPlanes* will verify that the data is IJK-ordered, change the Volume mode to I-planes, and cycle through the I-planes.

All of the example source code shown in this manual is included in the Tecplot distribution and is found in the **`adk/samples`** sub-directory below the Tecplot home directory.

AnimIPlanes uses source code files created by the **`CreateNewAddOn`** script (UNIX), or Tecplot Add-on Wizard (Windows).

Our project name will be "AnimIPlanes" and the add-on name will be "Animate I Planes."

When running **`CreateNewAddOn`** or Tecplot Add-on Wizard answer the questions as follows:

• Project Name (Base name	**`AnimIPlanes`**
• Add-on name:	Animate I Planes
• Company name:	[Your company name]
• Type of add-on:	General Purpose
• Language:	C
• Use TGB to create a platform-independent GUI?	Yes
• Add a menu call back to the Tecplot "Tools" menu?	Yes
• Menu text:	Animate I Planes
• Menu callback option:	Launch a modeless dialog
• Dialog title:	Animate I Planes

After running the **`CreateNewAddOn`** script or Tecplot Add-on Wizard you should have the following files:

```
guibld.c      guicb.cguidefs.cmain.c
ADDGLBL.h     GUIDEFS.hgui.lay
```

You will also have other files specific to your platform, but we will only modify those above. The purpose of each file will be explained in detail as we proceed.

Verify that you can compile your project add-on and load it into Tecplot. If you cannot, refer to Chapter 2 "Creating Add-ons under Windows," or Chapter 3 "Creating Add-ons under UNIX."

Windows users should select the Debug page in the Project/Settings menu and make the following entries:

`Executable for debug session:` *<path to Tecplot executable>*
`Working directory:`
`Program Arguments: Debug/AnimIPlanes.dll`

Now create your main dialog. This will be displayed when *Animate I Planes* is selected from Tecplot's Tools menu. The dialog will have two labels, one button, one text field, and a multi-selection list. You will be able to select a specific set of zones to animate from the list, specify a skip level in the text field, and clicking the button will perform the animation.

Before beginning, be sure that Tecplot GUI Builder (TGB) is available from Tecplot's Tools menu. If TGB is not available, do the following

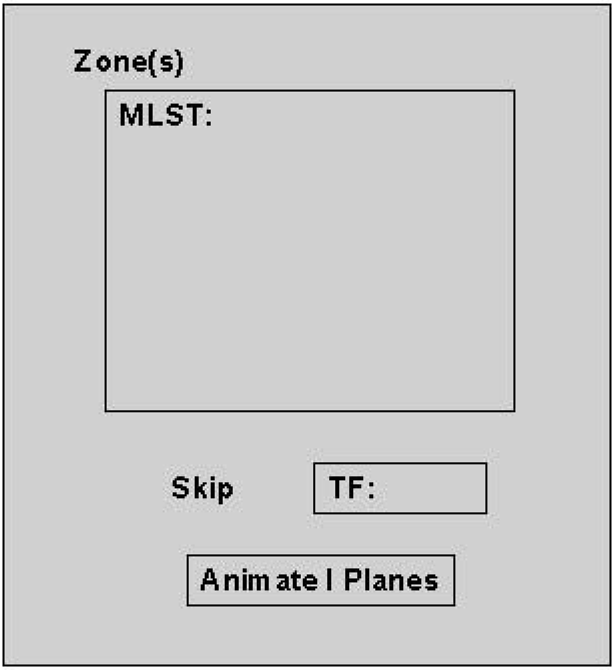
Windows

In the Tecplot Home Directory edit the file **`tecplot.add`** and add the line:
`# $!LoadAddOn "guibld"`

UNIX

Edit the file **`tecdev.add`** in your Add-on Development Root Directory and add the line:
`$!LoadAddOn "guibuild"`

Resize the frame and edit the layout as follows:

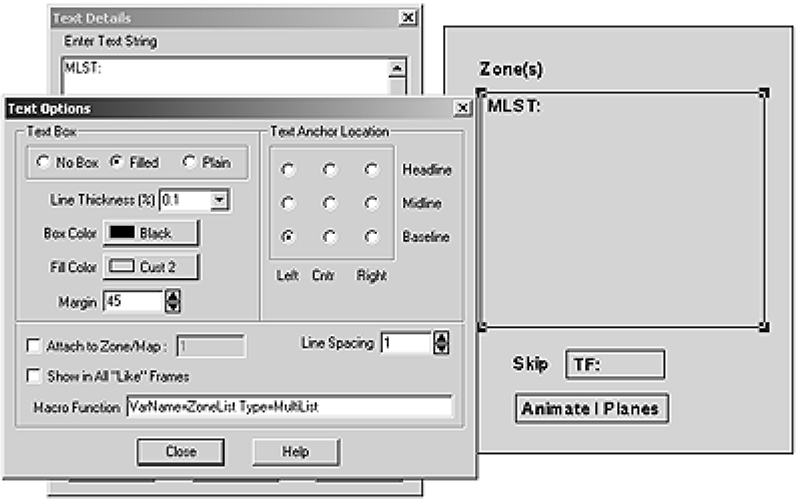


You can edit a control by clicking on it, then choosing Object Details and editing as you would text.

Note: Although the text fields and buttons are referred to as controls, since they exist in a layout file they are actually Tecplot text objects.

Double-click on the MLST: multi-selection list and select Options. In the Macro Function field, set **`VarName=ZoneList`**. This will be the base name of the callback associated with the multi-selection list. Also change the Macro Function for the TF: text field to **VarName=Skip**, and change the Macro Function for the Animate I Planes button to **VarName=AnimPlanes**.

The base names are truncated after 12 characters, so we specify a macro command for the button here.



Text Details

Enter Text String: MLST:

Text Options

Text Box: ○ No Box ● Filled ○ Plain

Line Thickness (%) 0.1

Box Color: Black

Fill Color: Cust 2

Margin 45

Text Anchor Location: Left / Baseline selected

□ Attach to Zone/Map : 1

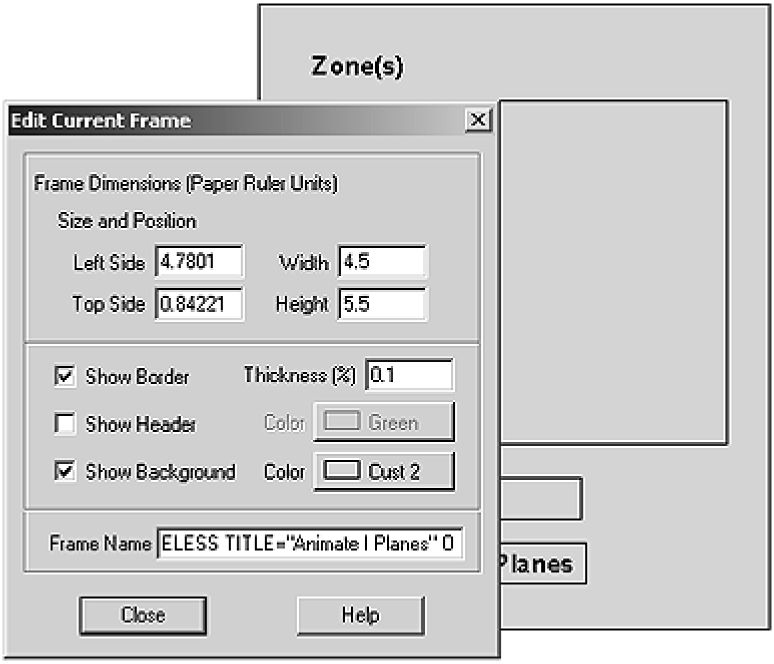
□ Show in All "Like" Frames

Line Spacing 1

Macro Function VarName=ZoneList Type=MultiList

Next, The dialog title is specified in the Edit Current Frame dialog. Double-click on the dialog frame and verify that the frame is:

```
ID=1 MODE=MODELESS TITLE="Animate I Planes"
```



Edit Current Frame

Frame Dimensions (Paper Ruler Units)

Size and Position

Left Side 4.7801 Width 4.5

Top Side 0.84221 Height 5.5

☑ Show Border Thickness (%) 0.1

□ Show Header Color Green

☑ Show Background Color Cust 2

Frame Name ELESS TITLE="Animate I Planes" 0

You can now build the source for this layout. From the TGB dialog click Go Build.

Rename the file **guicb.tmp** to be **guicb.c** (replacing the existing **guicb.c** with **guicb.tmp**).

When the dialog is launched we need to make sure that the **Skip** and **ZoneList** text fields are filled in properly. To initialize **Skip** we will define the skip to be a reasonable default value, and set it every time the dialog is launched. This initialization will take place in the **Dialog1Init_CB()** function. This function is called every time the dialog is launched.

Note the following line in **guicb.c**, just below the **#include** statements:

```
#define DEFAULT_SKIP “1”
```

and the following code used as the dialog initialization callback:

```
static void Dialog1Init_CB(void)
{
  TecUtilLockStart(AddOnID);
  /*<<< Add init code (if necessary) here>>>*/
  TecGUITextFieldSetString(Skip_TF_D1, DEFAULT_SKIP);
  TecUtilLockFinish(AddOnID);
}
```

To initialize **ZoneList** we will write a separate function, then call that function from the **Dialog1Init_CB()** function. This function will be called elsewhere in this exercise.

The following code is above the **InitTecAddOn()** function:

```
void FillZoneList(void)
{
  if (TecUtilDataSetIsAvailable())
    {
      EntIndex_t NumZones, i;

      TecUtilDataSetGetInfo(NULL, &NumZones, NULL);
      TecGUIListDeleteAllItems(ZoneList_MLST_D1);
      for (i = 1; i <= NumZones; i++)
        {
          char *ZoneName;
          TecUtilZoneGetName(i, &ZoneName);
          TecGUIListAppendItem(ZoneList_MLST_D1, ZoneName);
          TecUtilStringDealloc(&ZoneName);
        }
    }
  else
    TecGUIListDeleteAllItems(ZoneList_MLST_D1);
}
```

This function will fill the zone list with the zone names of the data set in the current frame. If there is no data set, the items in the list are deleted.

This function is called in the dialog initialization callback in **guicb.c.** The callback should now look like:

```
static void Dialog1Init_CB(void)
{
  TecUtilLockStart(AddOnID);
  /*<<< Add init code (if necessary) here>>>*/
  TecGUITextFieldSetString(Skip_TF_D1, DEFAULT_SKIP);
  FillZoneList();
  TecUtilLockFinish(AddOnID);
}
```

Since the function body of **FillZoneList()** is in **main.c**, add the following line to **ADDGLBL.h**:
EXTERN void FillZoneList(void);

THE ANIMATE I PLANES BUTTON

When the Animate I Planes button is clicked, we want to animate the I-planes. We will create a function called **AnimatePlanes()**, and add a call to that function in the **AnimatePlanes_BTN_D1_CB()** callback function.

Before calling the **AnimatePlanes()** function we need to collect data from the dialog and check to see that there is a data set available. The **AnimatePlanes()** function will take two parameters, **ZoneSet** and **Skip**. **ZoneSet** will contain the zones that were selected in the dialog, and **Skip** will be the skip value that was entered in the text field:

```
static void AnimPlanes_BTN_D1_CB(void)
{
  TecUtilLockStart(AddOnID);

  /* Make sure there is a dataset */
  if (TecUtilDataSetIsAvailable())
    {
      LgIndex_t  Count     = 0;
      LgIndex_t *Selection = NULL;
      Set_pa     ZoneSet   = TecUtilSetAlloc(TRUE);

      /* Get the Skip value from the text field */
      char *strSkip = TecGUITextFieldGetString(Skip_TF_D1);

      /* Get the selected zones from the ZoneList */
      TecGUIListGetSelectedItems(ZoneList_MLST_D1, &Selection, &Count);
      if (Count > 0)
        {
          LgIndex_t i;

          /* Put the selected items into ZoneSet */
          for (i = 0; i < Count; i++)
            TecUtilSetAddMember(ZoneSet, Selection[i], TRUE);

          TecUtilArrayDealloc((void **)&Selection);
        }
      /* Make sure a zone has been picked */
      if (ZoneSet != NULL) /* ...do the animation */
        AnimatePlanes(ZoneSet, atoi(strSkip));
      else
        TecUtilDialogErrMsg(“No zones have been picked.”);

      /* Deallocate the ZoneSet and strSkip string when we are done with them
*/
      if (ZoneSet != NULL)
        TecUtilSetDealloc(&ZoneSet);
      if (strSkip != NULL)
        TecUtilStringDealloc(&strSkip);
    }
  else
    TecUtilDialogErrMsg(“No data set available.”);

  TecUtilLockFinish(AddOnID);
}
```

We collect the information from the dialog and then pass that information off to **AnimatePlanes()** to carry out the animation. Because **ZoneSet** is initialized to **NULL**, we can tell if there were any selections. If there were not, we display an error message reading “No zones have been picked.”

WRITING THE ANIMATEPLANES() FUNCTION

This function will perform the actual animation. It takes two parameters, **ZoneSet** and **Skip**. These parameters are collected in the **AnimatePlanes** button callback function in **main.c**:

```
void AnimatePlanes(Set_pa ZoneSet,
                   int    Skip)
{
  LgIndex_t   MaxIndex = 0;
  EntIndex_t  CurZone;
  SetIndex_t  NumberOfZonesInSet;
  SetIndex_t  Index;
  Set_pa      IJKZoneSet = TecUtilSetAlloc(TRUE);
  char       *strMacroCommand;


  /* Get the number of zones in ZoneSet */
  NumberOfZonesInSet = TecUtilSetGetMemberCount(ZoneSet);

  if (TecUtilMacroIsRecordingActive() &&
      (NumberOfZonesInSet >= 1))
    {
      strMacroCommand = TecUtilStringAlloc(2000, “Macro Command”);
      strcpy(strMacroCommand, “ZONESET=”);
    }


  /*
```

```
   * Create a subset of ZoneSet that includes only
   * IJK Ordered Zones.  Do this by looping through
   * all the zones in ZoneSet, check to see if the zone
   * is IJK Ordered.  Then add the zone to IJKZoneSet
   */
  for (Index = 1; Index <= NumberOfZonesInSet; Index++)
    {
      /* Get the current zone */
      CurZone = (EntIndex_t)TecUtilSetGetMember(ZoneSet, Index);

      /* Make sure the current zone is enabled */
      if (TecUtilZoneIsEnabled(CurZone))
        {
          /* Only add the zone if it is IJK ordered */
          if (ZoneIsIJKOrdered(CurZone))
            {
              TecUtilSetAddMember(IJKZoneSet, CurZone, TRUE);
              /* Find the greatest IMax of all the valid IJK ordered zones */
              MaxIndex = MAX(MaxIndex, GetIMaxFromCurZone(CurZone));
            }

          if (TecUtilMacroIsRecordingActive())
            {
              sprintf(&strMacroCommand[strlen(strMacroCommand)], “%d”,
CurZone);
              if (Index != NumberOfZonesInSet)
                strcat(strMacroCommand, “,”);
            }
        }
    }


  /* Only proceed if there is at least one IJK ordered zone */
  if (TecUtilSetGetMemberCount(IJKZoneSet) >= 1)
    {
      Boolean_t IsOk = TRUE;

      /* Setup the zones for animation of I-Planes */

      /* Change the cell type to planes */
      TecUtilZoneSetIJKMode(SV_CELLTYPE,
                            NULL,
                            IJKZoneSet,
                            (ArbParam_t)IJKCellType_Planes);

      /* Display only the I-Planes */
      TecUtilZoneSetIJKMode(SV_PLANES,
                            NULL,
                            IJKZoneSet,
                            (ArbParam_t)Planes_I);

      /* Make sure that the Skip is greater than or equal to one. */
      if (Skip < 1)
        Skip = 1;

      /* Do the actual animation */
      TecUtilDoubleBuffer(DoubleBufferAction_On);
      for (Index = 1; IsOk && Index <=MaxIndex; Index += Skip)
        {
          /*
           * Set the range of the I-Planes so that the
           * minimum I-Plane to display is the same as
           * the maximum displayed.  Then increment
           * by Skip.  This will make the I-Planes “move”
           */
          TecUtilZoneSetIJKMode(SV_IRANGE,
                                SV_MIN,
                                IJKZoneSet,
                                (ArbParam_t)Index);
          TecUtilZoneSetIJKMode(SV_IRANGE,
                                SV_MAX,
                                IJKZoneSet,
                                (ArbParam_t)Index);
          IsOk = TecUtilRedraw(TRUE);
          TecUtilDoubleBuffer(DoubleBufferAction_Swap);
        }
      TecUtilDoubleBuffer(DoubleBufferAction_Off);

      if (IsOk && TecUtilMacroIsRecordingActive())
        {
          /* At this point we have all the IJK ordered zones.
           * So all we need to add is the skip value.  Add a semi-colon
           * to the end to signify the end of the IJKZoneSet information.
           */
          strcat(strMacroCommand, “; “);
          sprintf(&strMacroCommand[strlen(strMacroCommand)], “SKIP=%d”,
Skip);
          strMacroCommand[strlen(strMacroCommand)] = ‘\0’;

          /* Record the command */
          TecUtilMacroRecordAddOnCommand(“animiplanes”, strMacroCommand);
          TecUtilStringDealloc(&strMacroCommand);
        }
    }
  TecUtilSetDealloc(&IJKZoneSet);
}
```

Note the use of double buffering when we do the animation. If we do not double buffer, there will be a significant amount of flickering during animation. This is due to the time it takes to draw the other zones. There are a few functions called above that have not yet been defined; they check to see if the zone passed is IJK-ordered.

Note the following functions above the **`AnimatePlanes()`** function:

```
static Boolean_t ZoneIsIJKOrdered(EntIndex_t ZoneNum)
{
  Boolean_t IsOk;
  LgIndex_t IMax,JMax,KMax;

  TecUtilZoneGetInfo(ZoneNum,
                     &IMax,
                     &JMax,
                     &KMax,
                     NULL, /* XVar */
                     NULL, /* YVar */
                     NULL, /* ZVar */
                     NULL, /* NMap */
                     NULL, /* UVar */
                     NULL, /* VVar */
                     NULL, /* WVar */
                     NULL, /* BVar */
                     NULL, /* CVar */
                     NULL); /* SVar */
  IsOk = (IMax > 1 && JMax > 1 && KMax > 1);
  return IsOk;
}
```

This function is added for convenience, so as to not clutter **`AnimatePlanes()`**.

```
static LgIndex_t GetIMaxFromCurZone(EntIndex_t ZoneNum)
{
  LgIndex_t IMax;
  TecUtilZoneGetInfo(ZoneNum,
                     &IMax,
                     NULL, /* JMax */
                     NULL, /* KMax */
                     NULL, /* XVar */
                     NULL, /* YVar */
                     NULL, /* ZVar */
                     NULL, /* NMap */
                     NULL, /* UVar */
                     NULL, /* VVar */
                     NULL, /* WVar */
                     NULL, /* BVar */
                     NULL, /* CVar */
                     NULL); /* SVar */
  return IMax;
}
```

Compile the add-on and make sure that it runs properly. If you have two frames with different data sets, the zone list will not be updated when switching between frames.

Now we will add *functionality* to allow the zone list to update properly. To do this we will need to listen for state changes. When something in Tecplot changes, such as a new top frame, Tecplot broadcasts a message saying that there is a new top frame. We are going to add code to our add-on to allow it to listen for these messages. This is called a State Change Callback function.

During the setup of this add-on we requested to have state change monitoring code included in the initial build. This code was added to **`main.c`**. Now locate the function **`AnimIPlanesStateChangeCallback()`** in **`main.c`**. Notice that it already contains a switch statement with all the state changes you can monitor. The that the add-on is concerned about are grouped together in the state change callback:

```
case StateChange_NewTopFrame   :
case StateChange_ZonesAdded    :
case StateChange_ZonesDeleted :
case StateChange_FrameDeleted :
case StateChange_ZoneName      :
case StateChange_DataSetReset :
```

And a call to **`FillZoneList()`** is performed when these state changes are detected. The resulting code should look as follows:

```
void STDCALL AnimIPlanesStateChangeMonitor(StateChange_e StateChange,
                                           ArbParam_t    CallData)
{
  TecUtilLockStart(AddOnID);
  switch (StateChange)
    {
      case StateChange_NewTopFrame   :
      case StateChange_ZonesAdded    :
      case StateChange_ZonesDeleted :
      case StateChange_FrameDeleted :
      case StateChange_ZoneName      :
      case StateChange_DataSetReset :
        {
          /*
           * State changes may come in here while the dialog
           * is down.  We only want to fill the zone list
           * while the dialog is up.
           */
          if (TecGUIDialogIsUp(Dialog1Manager))
            FillZoneList();

        } break;
      default: break;
    }
  TecUtilLockFinish(AddOnID);
}
```

AnimIPlanes is now complete. Recompile and load into Tecplot.

EXERCISES

1. Currently there is nothing to inform users they have entered an invalid number for the skip, such as a negative number or zero. Add error checking in the text field callback to check for a valid positive integer.

2 Check that the integer in the text field is less than or equal to the maximum I-Max for the selected zones.

3 Allow the animation of J- and K-planes. Adding an option menu to the interface with the types of planes as options would be a good place to start.

4 Add code to make the add-on remember the last skip value entered, such that when the dialog is closed and reopened the last skip value is the default in the text field.

5 Allow input of start and end planes. This would allow animation from a larger plane index to a smaller index, and allow a specific range of planes to animate.

CHAPTER 12 The Polynomial Integer Add-on

INTRODUCTION TO THE POLYINT EXTENDED CURVE-FIT

PolyInt, the Tecplot add-on you will build in this tutorial, is an example of an extended curve-fit add-on that does not have any settings which may be configured. This add-on will add an option to the single selection list that is launched by the Curve Type/Extended option on the Curves page of the Mapping Style dialog.

This add-on will perform three operations. The only required operation is to calculate the curve-fit of discrete XY-data. The second operation is to supply Tecplot with a dependent value when the plot is probed. The third is to present a string to the XY Plot Curve Info dialog.

Note: For the purposes of this tutorial, it is assumed that you have already read the chapters “Creating Add-ons Under Windows” and/or “Creating Add-ons Under UNIX” in the *ADK User’s Manual*, and that you have successfully created and compiled a set of starter files. All of the code from this point on is platform-independent, and you can work through the tutorial using either a Windows or UNIX environment.

All of the example of source code shown in this manual is included in the Tecplot distribution and are found in the **`adk/samples`** sub-directory below the Tecplot home directory.

GETTING STARTED

PolyInt will use the following source code files. Each one will be automatically created by the **`CreateNewAddOn`** script (UNIX) or the Tecplot Add-on Wizard (Windows). The project name and the add-on name will both be PolyInt.

When running **`CreateNewAddOn`** or the Tecplot 10 Add-on Wizard, answer the questions as follows:

• Project Name (Base name):	PolyInt
• Add-on name:	PolyInt
• Company Name:	[Your company name]
• Type of add-on:	Extended Curve-Fit
• Language:	C
• Allow Configurable Settings:	No
• Create callback function for more accurate probing:	Yes

After running the CreateNewAddOn script or the Tecplot 10 Add-on Wizard, you should have the following files:

```
engine.c     main.c
ADDGLBL.h    ENGINE.h
```

You will also have other files specific to your platform, but the files above are the only ones we will be modifying. The purpose of each file will be explained in detail as we proceed through the tutorial.

At this point, you should verify that you can compile your add-on and load it into Tecplot.

If you are unable to compile or load your add-on, we recommend that you refer to Chapter 2 “Creating Add-ons Under UNIX” or Chapter 3, “Creating Add-ons Under Windows” in the *ADK User's Manual* before proceeding.

SOURCE FILES

Since this add-on has no dialog, we will only be dealing with four files:

main.c, **engine.c**, **ADDGLBL.h** and **ENGINE.h**.

File main.c

This file contains the add-on registration routine. If you open the file, you will see a call to **TecUtilCurveRegisterExtCrvFit**. It is this function that registers the extended curve-fit add-on with Tecplot. In **main.c**, the call to **TecUtilCurveRegisterExtCrvFit** should appear as follows:

```
      TecUtilCurveRegisterExtCrvFit(ADDON_NAME,
                                    XYDataPointsCallback,
                                    ProbeValueCallback,
                                    CurveInfoStringCallback,
                                    NULL,  /* CurveSettingsCallback */
                                    NULL); /* AbbreviatedSettingsString-
Callback */
```

Notice that parameters five and six are **NULL**. This is because this add-on has no settings which may be configured.

Since the extended curve-fit feature is unique to Version 9.0 and later, notice that the **InitTecAddOn()** function contains version checking. This ensures that previous versions of Tecplot cannot load extended curve-fit add-ons.

We will define the three registered callbacks in **engine.c** and prototype them in **ENGINE.h**.

File ENGINE.h

Open **ENGINE.h** and verify that the following lines exist:

```
extern Boolean_t STDCALL XYDataPointsCallback(FieldData_pa RawIndV,
                                               FieldData_pa RawDepV,
                                             CoordScale_e IndVCoordScale,
                                             CoordScale_e DepVCoordScale,
                                               LgIndex_t    NumRawPts,
                                               LgIndex_t    NumCurvePts,
                                               EntIndex_t   XYMapNum,
                                              char         *CurveSettings,
                                             double       *IndCurveValues,
                                            double       *DepCurveValues);
extern Boolean_t STDCALL CurveInfoStringCallback(FieldData_pa  RawIndV,
                                                  FieldData_pa  RawDepV,
                                                  CoordScale_e
 IndVCoordScale,
                                                  CoordScale_e
 DepVCoordScale,
                                                 LgIndex_t     NumRawPts,
                                                  EntIndex_t    XYMapNum,
                                              char        *CurveSettings,
                                                  char        **CurveIn-
 foString);
 extern Boolean_t STDCALL ProbeValueCallback(FieldData_pa RawIndV,
                                             FieldData_pa RawDepV,
                                             CoordScale_e IndVCoordScale,
                                             CoordScale_e DepVCoordScale,
                                             LgIndex_t    NumRawPts,
                                             LgIndex_t    NumCurvePts,
                                             EntIndex_t   XYMapNum,
                                             char        *CurveSettings,
                                             double       ProbeIndValue,
                                             double      *ProbeDepValue);
```

Each of these functions will be defined in **`engine.c`**.

engine.c

When the source files are created, they are filled with code that will compute a simple average of the dependent values. This code is not needed for this add-on and should be deleted. Delete the **`SimpleAverage()`** function and all of the code in the callback functions (do not delete the function declarations themselves).

In **`engine.c`** we will define the three callbacks that are prototyped above. First we will deal with the function that actually performs the curve-fit. The function is called **`PolyInt()`**. It is based on a method given in the Stineman article from *Creative Computing* (July, 1980). Much of this tutorial will focus on manipulating the data into a form that the **`PolyInt()`** function can use. The algorithm used here will not be explained since it is beyond the scope of this tutorial.

The **`PolyInt()`** function takes an array that we call **`Data`** and some information about the contents of the array. The **`Data`** array is separated into four separate blocks.

- **Block 1:** Raw independent data values.
- **Block 2:** Raw dependent data values.
- **Block 3:** Calculated independent values (based on the number of points on the calculated curve).
- **Block 4:** Calculated dependent values (to be filled in by **`PolyInt()`** function).

We will also pass the indices of the start of each block, the number of raw data points, and the number of points on the calculated curve to the **`PolyInt()`** function.

Note the following code in engine.c just below the last **`#include`** statement:

```
/**
 * Interpolate y=f(x) using the method given in Stineman article from
 * Creative Computing (July 1980). At least 3 points required for
 * interpolation, if fewer then use linear interpolation...
 *
 * Data is treated as a 1 based array, while lx,ly,lxn,lyn are treated as
0
 * base.
 *
 * @param npts
 *     number of original data points
 * @param lx
 *     location of x data points
 * @param ly
 *     location of y data points
 * @param nptn
 *     number of points on the fitted curve
 * @param lxn
 *     location of fitted x points
 * @param lyn
 *     location of fitted y points
 * @param data
 *     working array
 */
void PolyInt(int     npts,
             int     lx,
             int     ly,
             int     nptn,
             int     lxn,
             int     lyn,
             double *data)
{
  int    j,j1,i,ix,jx,kx,ixx,jxx;
  double xv,yv,dydx,dydx1,s,y0,dyj,dyj1;

  j  = 1;
  j1 = j+1;

  /* Isolate the data(lx+j) and the data(lx+j+1) that bracket xv... */
  for (i=1; i<=nptn; i++)
    {
      xv = data[lxn+i];
      while (xv > data[lx+j1])
        {
          j++;
          j1 = j+1;
        }

      if (npts == 1)
        yv = data[ly+j];

      if (npts == 2)
        yv = data[ly+2]-(data[lx+j1]-xv)*(data[ly+j1]-data[ly+j])/
            (data[lx+j1]-data[lx+j]);

      if (npts >= 3)
        {
          /*
          * Calculate the slope at the jth point (from fitting a circle
thru
           * 3 points and getting slope of circle).
           */
          ix = 1;
          jx = 2;
          kx = 3;
          if (j != 1)
            {
              ix = j-1;
              jx = j;
              kx = j+1;
            }

          dydx = (((data[ly+jx]-data[ly+ix])*
                   (pow(data[lx+kx]-data[lx+jx],2)+
                    pow(data[ly+kx]-data[ly+jx],2))+
                   (data[ly+kx]-data[ly+jx])*
                   (pow(data[lx+jx]-data[lx+ix],2)+
                    pow(data[ly+jx]-data[ly+ix],2)))/
                  ((data[lx+jx]-data[lx+ix])*
                   (pow(data[lx+kx]-data[lx+jx],2)+
                    pow(data[ly+kx]-data[ly+jx],2))+
                   (data[lx+kx]-data[lx+jx])*
                   (pow(data[lx+jx]-data[lx+ix],2)+
                    pow(data[ly+jx]-data[ly+ix],2))));
          if (j == 1)
            {
              ixx = ix;
              jxx = jx;
              s = (data[ly+jxx]-data[ly+ixx])/(data[lx+jxx]-
data[lx+ixx]);

              if (s != 0.0)
                {
                 if (!((s >= 0.0 && s > dydx) || (s <= 0.0 && s < dydx)))
                    dydx = s+(fabs(s)*(s-dydx))/(fabs(s)+fabs(s-dydx));
                  else
                    dydx = 2.0*s-dydx;
                }
            }

          /* Calculate the slope at j+1 point. */
          ix = nptn-2;
          jx = nptn-1;
          kx = nptn;

          if (j1 != nptn)
            {
              ix = j1-1;
              jx = j1;
              kx = j1+1;
            }
          dydx1 = (((data[ly+jx]-data[ly+ix])*
                    (pow(data[lx+kx]-data[lx+jx],2.)+
                     pow(data[ly+kx]-data[ly+jx],2.))+
                    (data[ly+kx]-data[ly+jx])*
                    (pow(data[lx+jx]-data[lx+ix],2.)+
                     pow(data[ly+jx]-data[ly+ix],2.)))/
                   ((data[lx+jx]-data[lx+ix])*
                    (pow(data[lx+kx]-data[lx+jx],2.)+
                     pow(data[ly+kx]-data[ly+jx],2.))+
                    (data[lx+kx]-data[lx+jx])*
                    (pow(data[lx+jx]-data[lx+ix],2.)+
                     pow(data[ly+jx]-data[ly+ix],2.))));

          if (j1 == nptn)
            {
              ixx = jx;
              jxx = kx;
              s = (data[ly+jxx]-data[ly+ixx])/
                  (data[lx+jxx]-data[lx+ixx]);
              if (s != 0.0)
                {
                  if (!((s >= 0.0 && s > dydx1) ||
                        (s <= 0.0 && s < dydx1)))
                    dydx1 = s+(fabs(s)*(s-dydx1))/(fabs(s)+fabs(s-
dydx1));

                  else
                    dydx1 = 2.0*s-dydx1;
                }
            }

          /*
           * Calculate s=slope between j and j+1 points
           * y0   = y-value if linear interp used
           * dyj  = delta-y at the j-th point
           * dyj1 = delta-y at the j+1 point
           */
          s    = (data[ly+j1]-data[ly+j])/(data[lx+j1]-data[lx+j]);
          y0   = data[ly+j]+s*(xv-data[lx+j]);
          dyj  = data[ly+j]+dydx*(xv-data[lx+j])-y0;
          dyj1 = data[ly+j1]+dydx1*(xv-data[lx+j1])-y0;

          /* Calculate y... */

          if (dyj*dyj1 == 0.0)
            yv = y0;
          if (dyj*dyj1 > 0.0)
            yv = y0+(dyj*dyj1)/(dyj+dyj1);
          if (dyj*dyj1 < 0.0)
            yv = y0+((dyj*dyj1*(xv-data[lx+j]+xv-data[lx+j1]))/
                     ((dyj-dyj1)*(data[lx+j1]-data[lx+j])));
```

```
        }

      data[lyn+i] = yv;
    }
}
```

THE XYDATAPOINTSCALLBACK() FUNCTION

Knowing that the **PolyInt()** function uses a single array containing all the raw and calculated independent values, we must prepare this array in the **XYDataPointsCallback** function and pass it on to the **PolyInt()** function. Once the array is passed on to **PolyInt()**, it will be returned with the calculated points filled in, at which time we must extract those points from the working array and place them into the array that Tecplot passed to the **XYDataPointsCallback()** function.

See **TecUtilCurveRegisterExtCrvFit()** in the *ADK Reference Manual* for an explanation of the parameters of this function.

The **XYDataPointsCallback()** has the following structure:

1. Allocate and initialize the working array, called **Data**.

2. Fill the working array, **Data**, with the raw data and the calculated independent values.

3. Pass the working array, **Data**, to the **PolyInt()** function. This will fill in the calculated dependent values.

4. Extract the data from the working array, **Data**, and place into the arrays that Tecplot passed in.

5. Free the working array.

The code for the XYDataPointsCallback() is below:

```
Boolean_t STDCALL XYDataPointsCallback(FieldData_pa  RawIndV,
                                       FieldData_pa  RawDepV,
                                       CoordScale_e  IndVCoordScale,
                                       CoordScale_e  DepVCoordScale,
                                       LgIndex_t     NumRawPts,
                                       LgIndex_t     NumCurvePts,
                                       EntIndex_t    XYMapNum,
                                       char         *CurveSettings,
                                       double       *IndCurveValues,
                                       double       *DepCurveValues)
{
  Boolean_t IsOk = TRUE;
  int       ii;
  double    *Data = NULL;
  int       TotalNumDataPts;

  TecUtilLockStart(AddOnID);

  /*
   * Data will contain all the data points and is 1 base:
   *   RawIndpts
   *   RawDepPts
   *   IndCurveValues
   *   DepCurveValues
   * Therefore, the array must be large enough to
   * contain all these points: 2*(NumRawPts+NumCurvePts).
   */
  TotalNumDataPts = 2*(NumRawPts+NumCurvePts);
  Data = malloc((TotalNumDataPts+1)*sizeof(double));
  if (Data != NULL)
    {
      /* Initialize Data to contain all zero. */
      for (ii = 0; ii < TotalNumDataPts+1; ii++)
        Data[ii] = 0;
    }
  else
    IsOk = FALSE;

  if (IsOk)
    {
      int lx;
      int ly;
      int lxn;
      int lyn;

      /* Setup the working array, Data. */
      PrepareWorkingArray(RawIndV,
                          RawDepV,
                          NumRawPts,
                          NumCurvePts,
                          &lx,
                          &ly,
                          &lxn,
                          &lyn,
                          Data);
      /* Perform the curve fit. */
      PolyInt(NumRawPts,
              lx,
              ly,
              NumCurvePts,
              lxn,
              lyn,
              Data);

     /* Extract the values from Data that were placed there by the curve
fit. */
      ExtractCurveValuesFromWorkingArray(NumCurvePts,
                                         lxn,
                                         lyn,
                                         Data,
                                         IndCurveValues,
                                         DepCurveValues);
      free(Data);
    }
  TecUtilLockFinish(AddOnID);
  return IsOk;
}
```

Notice that in this function, the CurveSettings and XYMapNum variables are never referenced. This is because there are no settings which may be configured for this curve-fit. The only information in this function that is required by Tecplot is the return value (TRUE or FALSE), and that the IndCurveValues and DepCurveValues arrays are filled. Tecplot will plot whatever values are placed in these arrays. If the values do not make sense, the resulting plot will not make sense. The burden is on the add-on writer to make sure that the values placed in these arrays are correct.

Also, notice that there are two functions we have referenced that must still be written. These functions take care of steps 2 and 4 as outlined in the function structure above.

THE PREPAREWORKINGARRAY() FUNCTION

This function will fill the working array, `Data`, with the raw data and the calculated independent curve points. It will also return the indices within the `Data` array to the different blocks of data. As stated above:

- **lx:** Start of the raw Independent data.
- **ly:** Start of the raw Dependent data.
- **lxn:** Start of the calculated independent data.
- **lyn:** Start of the calculated dependent data.

Note the following function above the `XYDataPointsCallback()` function.

```
static void PrepareWorkingArray(FieldData_pa  RawIndV,
                                FieldData_pa  RawDepV,
                                LgIndex_t     NumRawPts,
                                LgIndex_t     NumCurvePts,
                                int          *lx,
                                int          *ly,
                                int          *lxn,
                                int          *lyn,
                                double       *Data)
{
  double FirstValidPoint;
  double LastValidPoint;
  double StepSize;
  int    ii;

  /*
   * The followint are indices to start points of
   * the data blocks in the 1 based arrray, Data
   *   lx  - Start of the raw Independent data.
   *   ly  - Start of the raw  Dependent data.
   *   lxn - Start of the calculated independent data.
   *   lyn - Start of the calculated   dependent data.
   *
   * The PolyInt function treats lx,ly,lxn,lyn as 0 base
   * indices, but treats Data as a 1 base array.
   */
  *lx = 0;
  *ly = NumRawPts;
  *lxn = 2*NumRawPts;
  *lyn = 2*NumRawPts+NumCurvePts;

  /* Fill the first blocks of the Data array with the Raw Data Values. */
  for (ii = 1; ii <= NumRawPts; ii++)
    {
      Data[*lx+ii] = TecUtilDataValueGetByRef(RawIndV, ii);
      Data[*ly+ii] = TecUtilDataValueGetByRef(RawDepV, ii);
    }

  /*
   * Calculate the size of steps to take while stepping
   * along the independent variable range.
   */
  TecUtilDataValueGetMinMaxByRef(RawIndV,
                                 &FirstValidPoint,
                                 &LastValidPoint);
  StepSize = (LastValidPoint-FirstValidPoint)/(NumCurvePts-1);

  /*
   * Fill the third block of the Data array with the
   * calculated independent values.
   */
  for (ii = 1; ii <= NumCurvePts; ii++)
    {
      double IndV = FirstValidPoint + (ii-1)*StepSize;
      if (IndV > LastValidPoint)
        IndV = LastValidPoint;
      Data[*lxn+ii] = IndV;
    }
}
```

THE EXTRACTCURVEVALUESFROMWORKINGARRAY() FUNCTION

This function will extract the calculated data from the working array, **Data**, and place it in the arrays that were passed to the **XYDataPointsCallback()** function by Tecplot. Tecplot will then use these values to plot the curve.

Note the following function above the **XYDataPointsCallback()** function.

```
static void ExtractCurveValuesFromWorkingArray(LgIndex_t  NumCurvePts,
                                               int         lxn,
                                               int         lyn,
                                               double     *Data,
                                              double     *IndCurveValues,
                                              double     *DepCurveValues)
{
  int ii;
  for (ii = 1; ii <= NumCurvePts; ii++)
```

```
    {
      IndCurveValues[ii-1] = Data[lxn+ii];
      DepCurveValues[ii-1] = Data[lyn+ii];
    }
}
```

At this point you should compile the add-on and load it into Tecplot. The curve-fit add-on is complete at this point, however there is other functionality that may be added. In the following sections we will add the probe value callback, and the curve information callback.

THE PROBEVALUECALLBACK() FUNCTION

The **ProbeValueCallback()** function is not required since Tecplot will perform a linear interpolation on the points that your curve-fit returns. However, if you have very few points in your curve, the value returned by Tecplot's built-in Probe function will return a value that is not on the actual curve, but on the approximated curve.

To avoid this problem, we will write the **ProbeValueCallback**. This callback will return a value that is actually calculated by your curve-fit. The method we use for this particular curve-fit is outlined below:

The **ProbeValueCallback** has the following structure:

1. Check that the probed independent value is within the bounds of the raw data.

2. If the number of curve points approximating the curve is small, reassign the number of points approximating the curve to be larger.

3. Allocate and initialize the working array, called **Data**.

4. Fill the working array with the raw data and the calculated independent values.

5. Insert the probed independent value into the working array, so a curve-fit is done at the actual probed independent value. Save the relative location of this value within the working array.

6. Pass the working array to the **PolyInt()** function. This will fill in the calculated dependent values.

7. Extract the probed dependent value from the working array, using the relative location saved in step 5.

8. Free the working array.

Note the following code in **engine.c**:

```
/**
 */
#define NUMPTSFORPROBING 3000

/**
 * This functions follows a similar process as the XYDataPointsCallback,
 * except it manually inserts ProbeIndValue in the list of the indepen-
dent
 * curve points.  It stores the index in the Data array for that value
and
 * uses that relative location to find the calculated ProbeDepValue.
 */
Boolean_t STDCALL ProbeValueCallback(FieldData_pa RawIndV,
                                     FieldData_pa RawDepV,
                                     CoordScale_e IndVCoordScale,
                                     CoordScale_e DepVCoordScale,
                                     LgIndex_t    NumRawPts,
                                     LgIndex_t    NumCurvePts,
                                     EntIndex_t   XYMapNum,
                                     char        *CurveSettings,
                                     double       ProbeIndValue,
                                     double      *ProbeDepValue)
{
  Boolean_t IsOk = TRUE;
  int       ii;
  double    FirstValidPoint;
  double    LastValidPoint;
  double    *Data = NULL;
  int       TotalNumDataPts;

  TecUtilLockStart(AddOnID);

  /* Make sure the probe is within the bounds of the data. */
  TecUtilDataValueGetMinMaxByRef(RawIndV,
                                 &FirstValidPoint,
                                 &LastValidPoint);
  IsOk = (ProbeIndValue >= FirstValidPoint &&
          ProbeIndValue <=  LastValidPoint);

  if (IsOk)
    {
      /*
       * If the Curve has too few points, crank the number of points
       * on the curve up, so we get a good approximation of the curve.
       */
      NumCurvePts = MAX(NUMPTSFORPROBING, NumCurvePts);

      TotalNumDataPts = 2*(NumRawPts+NumCurvePts);
      Data = malloc((TotalNumDataPts+1)*sizeof(double));
      if (Data != NULL)
        {
          /* Initialize Data to contain all zero. */
          for (ii = 0; ii < TotalNumDataPts+1; ii++)
            Data[ii] = 0;
        }
      else
        IsOk = FALSE;
    }

  if (IsOk)
    {
      int lx,ly,lxn,lyn;
      int ProbeValueIndex = -1;

      PrepareWorkingArray(RawIndV,
                          RawDepV,
                          NumRawPts,
                          NumCurvePts,
                          &lx,
                          &ly,
                          &lxn,
                          &lyn,
                          Data);
      IsOk = InsertProbeValueInWorkingArray(ProbeIndValue,
                                            NumCurvePts,
                                            lxn,
                                            &ProbeValueIndex,
                                            Data);
      if (IsOk && ProbeValueIndex != -1)
        {
          /* Perform the curve fit. */
          PolyInt(NumRawPts,
                  lx,
                  ly,
                  NumCurvePts,
                  lxn,
                  lyn,
                  Data);
          /* The dependent value is in the same relative location. */
          /* as the probed independent value. */
          *ProbeDepValue = Data[lyn+ProbeValueIndex];
        }
    }
  if (Data != NULL)
    free(Data);
  TecUtilLockFinish(AddOnID);
  return IsOk;
}
```

THE INSERTPROBEVALUEINWORKINGARRAY() FUNCTION

This function inserts the probed independent value into the working array so that the curve-fit will be performed exactly at the probed value. This is done by marching through the calculated independent values, and when two values that surround the probed value are found, the probed value replaces the lesser of the two surrounding values in the working array. Also, the relative location of the probed value is saved, so that the calculated dependent value can be extracted from the working array.

Note the following code above the **ProbeValueCallback()** in **engine.c**:

```
static Boolean_t InsertProbeValueInWorkingArray(double     ProbeInd-
Value,
                                                LgIndex_t  NumCurvePts,
                                                int        lxn,
                                             int       *ProbeValueIndex,
                                                double    *Data)
{
  Boolean_t Found = FALSE;
  int       ii;
  for (ii = 1; ii < NumCurvePts; ii++)
    {
      /* If the probed value is between the data points record its loca-
tion. */
      if (ProbeIndValue >= Data[lxn+ii] &&
          ProbeIndValue <= Data[lxn+ii+1])
        {
          *ProbeValueIndex = ii;
          Data[lxn+ii] = ProbeIndValue;
          Found = TRUE;
          break;
        }
    }
  return Found;
}
```

Compile and load the add-on into Tecplot. Now, you should be able to probe and have a real curve value be returned rather than the linear interpolation computed by Tecplot.

THE CURVEINFOSTRINGCALLBACK() FUNCTION

The **CurveInfoStringCallback()** function will pass a string to the XY-Plot Curve Info dialog. This string can be any information you wish to present to the dialog. Typical information in this dialogs are the curve coefficients. Since it is beyond the scope of this tutorial to calculate the coefficients of the curve, we will simply present a string to the dialog.

Examine the following code in **engine.c**:

```
Boolean_t STDCALL CurveInfoStringCallback(FieldData_pa RawIndV,
                                          FieldData_pa RawDepV,
                                          CoordScale_e IndVCoordScale,
                                          CoordScale_e DepVCoordScale,
                                          LgIndex_t    NumRawPts,
                                          EntIndex_t   XYMapNum,
                                          char        *CurveSettings,
                                          char       **CurveInfoString)
{
  Boolean_t IsOk = TRUE;

  *CurveInfoString = TecUtilStringAlloc(1000, "CurveInfoString");
  strcpy(*CurveInfoString, "Information about the curve goes here.\n");
  strcat(*CurveInfoString, "Such as curve coefficients.");
  return IsOk;
}
```

Again, compile and load the add-on into Tecplot. Upon running Tecplot, load rainfall.plt and change the curve type to Extended/PolyInt. Now, call up the XY-Plot Curve Info dialog. Notice that the string we added is now in the dialog.

As an exercise, add error messages to the **`XYDataPointsCallback()`** and the **`ProbeValueCallback()`** functions if they end up returning **`FALSE`**. This will inform the user that there was an error.

CHAPTER 13 The Simple Average Add-on

INTRODUCTION TO THE SIMPAVG EXTENDED CURVE-FIT

SimpAvg, the Tecplot add-on you will build in this tutorial, is an example of an extended curve-fit add-on that has settings which may be configured. The setting that we will be configuring in this add-on is the independent variable range. This curve-fit add-on will compute the average of the data within the specified independent variable range.

Note: For the purposes of this tutorial, it is assumed that you have already read the chapters “Creating Add-ons Under Windows” and/or “Creating Add-ons Under UNIX” in the *ADK User’s Manual*, and that you have successfully created and compiled a set of starter files. All of the code from this point on is platform-independent, and you can work through the tutorial using either a Windows or UNIX environment.

It is also assumed that you have created an add-on that has a dialog. If you have not done so, see Chapter 5, “The Equate Add-on.”

GETTING STARTED

SimpAvg will use the following source code files. Each one will be automatically created by the **`CreateNewAddOn`** script (UNIX) or the Tecplot Add-on Wizard (Windows). The project and add-on names will both be SimpAvg.

When running **`CreateNewAddOn`** or the Tecplot Add-on Wizard, answer the questions as follows:

• Project Name (Base name):	**`SimpAvg`**
• Add-on Name:	SimpAvg
• Company Name:	[Your company name]
• Type of Add-on:	Extended Curve-Fit
• Language:	C
• Allow Configurable Settings:	Yes
• Create Callback Function for More Accurate Probing:	No

After running the **`CreateNewAddOn`** script or Tecplot Add-on Wizard, you should have the following files:

`engine.c`	**`main.c`**	**`guibld.c`**	**`guicb.c`**
`guidefs.c`	**`ADDGLBL.h`**	**`ENGINE.h`**	**`GUIDEFS.h`**

You will also have other files specific to your platform, but the files above are the only ones we will be dealing with. The purpose of each file will be explained in detail as we proceed through the tutorial.

At this point, you should verify that you can compile your add-on and load it into Tecplot.

If you are unable to compile or load your add-on, we recommend that you refer to Chapter 2, “Creating Add-ons Under UNIX,” or Chapter 3, “Creating Add-ons Under Windows,” in the *ADK User’s Manual* before proceeding.

DESIGNING THE ADD-ON

Since this curve-fit will have settings which may be configured, we will need to make some decisions before writing the add-on.

What are the settings going to be?

- Use an Independent Variable Range.
- What is the IndVarMin?
- What is the IndVarMax?

What are the default settings?

- UseIndVarRange: **`FALSE`**.
- IndVarMin: **`-LARGEDOUBLE`** (-1E+150).
- IndVarMax: **`LARGEDOUBLE`** (1E+150).

What is the syntax for the `CurveSettings` string?

- Newline delimited (spaces delimiting the ‘=’ are required).
- Example:

```
UseIndVarRange = TRUE\n
IndVarMin = 2\n
IndVarMax = 7\n
```

How to maintain the values of the settings?

- The settings string will be maintained by Tecplot; however, we will create a struct as follows to hold the values that are contained in the settings string.

```
typedef struct
  {
    Boolean_t UseIndVarRange;
    double    IndVarMin;
    double    IndVarMax;
  } CurveParams_s;
```

- This structure will be placed in **`ADDGLBL.h`**.

HANDLING THE CURVESETTINGS STRING

The first thing we will do is lay some groundwork for how to handle the **`CurveSettings`** string. Since this string is maintained by Tecplot and is updated by the add-on, our add-on must know how to parse the string.

We will start with the function that creates the string. This function will make use of the **CurveParams_s** structure. If you have not done so already, add the **CurveParams_s** structure, as defined above, to **ADDGLBL.h**.

Now that the **CurveParams_s** structure is in place, we will create a function in **engine.c** called **CreateCurveSettingsString**. This function will take one parameter, the **CurveParams_s** structure, and return a string based on the values of the structure. The function is written as follows:

Add this prototype to **ENGINE.h**:
char *CreateCurveSettingsString(CurveParams_s CurveParams);

The following code is in **engine.c**:

```
/**
 * Creates a CurveSettings string based on the values
 * in the CurveParams structure that is passed in.
 */
char *CreateCurveSettingsString(CurveParams_s CurveParams)
{
  char S[1000];
  char *CurveSettings;

  if (CurveParams.UseIndVarRange)
    strcpy(S,"UseIndVarRange = TRUE\n");
  else
    strcpy(S,"UseIndVarRange = FALSE\n");

  sprintf(&S[strlen(S)], "IndVarMin = %G\n", CurveParams.IndVarMin);
  sprintf(&S[strlen(S)], "IndVarMax = %G\n", CurveParams.IndVarMax);

  S[strlen(S)] = '\0';
  CurveSettings = TecUtilStringAlloc(strlen(S), "CurveSettings");
  strcpy(CurveSettings, S);
  return CurveSettings;
}
```

Notice that this function calls **TecUtilStringAlloc**. The calling function is responsible for de-allocating the string returned by **CreateCurveSettingsString**. Also notice that the string is newline delimited as discussed above.

Now that we have a function that creates the **CurveSettings** string, create a function that will parse the newline delimited string and populate the **CurveParams_s** structure. The function that we will be writing will use several convenience functions that are defined in **adkutil.c**. This module can be found in **ADK/Samples/StateChg**. Copy the files **adkutil.c** and **ADKUTIL.h** to your add-on development directory. UNIX users must add these files to the Makefile, Windows users must add these files to the project. Once these files have been added, make sure you add the line:
#include "ADKUTIL.h"

at the top of **engine.c**.

The function that parses the **CurveSettings** string will take three parameters. The first is **XYMapNum**, which is the XY-map that is currently being operated on. The second is the **CurveSettings** string. The third is a pointer to the **CurveParams_s** structure. This function will not only parse the **CurveSettings** string, but also repair the string if the syntax is incorrect.

The following code is in **engine.c**:

```
/**
 * This function makes use of functions found in the
 * adkutil.c module to parse the CurveSettings string.
 */
void GetValuesFromCurveSettings(EntIndex_t    XYMapNum,
                                char          *CurveSettings,
                                CurveParams_s *CurveParams)
{
  Boolean_t  IsOk = TRUE;
# define     MAXCHARS 50
  char       Command[MAXCHARS+1];
  char       ValueString[MAXCHARS+1];
  char      *CPtr;
  char      *ErrMsg = NULL;

  if (CurveSettings != NULL && strlen(CurveSettings) > 0)
    {
      CPtr = CurveSettings;
      while (IsOk && *CPtr)
        {
          if (GetArgPair(&CPtr,
                         Command,
                         ValueString,
                         MAXCHARS,
                         &ErrMsg))
            {
              if (Str_ustrcmp(Command, “USEINDVARRANGE”) == 0)
                {
                  Boolean_t UseRange;
                  IsOk = Macro_GetBooleanArg(Command,
                                             ValueString,
                                             &UseRange,
                                             &ErrMsg);
                  if (IsOk)
                    CurveParams->UseIndVarRange = UseRange;
                }
              else if (Str_ustrcmp(Command, “INDVARMIN”) == 0)
                {
                  double Min;
                  IsOk = Macro_GetDoubleArg(Command,
                                            ValueString,
                                            -LARGEDOUBLE,
                                            LARGEDOUBLE,
                                            &Min,
                                            &ErrMsg);
                  if (IsOk)
                    CurveParams->IndVarMin = Min;
                }
              else if (Str_ustrcmp(Command, “INDVARMAX”) == 0)
                {
                  double Max;
                  IsOk = Macro_GetDoubleArg(Command,
                                            ValueString,
                                            -LARGEDOUBLE,
                                            LARGEDOUBLE,
                                            &Max,
                                            &ErrMsg);
                  if (IsOk)
                    CurveParams->IndVarMax = Max;
                }
              else
                {
                  ErrMsg = TecUtilStringAlloc((strlen(Command)+100),
                                              “error message”);
                  sprintf(ErrMsg, “Unknown argument: %s.”, Command);
                  IsOk = FALSE;
                }
            }
          else /* GetArgPair Failed. */
            IsOk = FALSE;
        }
    }
  else /* CurveSettings is an invalid string. */
    IsOk = FALSE;

  /* Repair the string.  Display the Error Message if needed. */
  if (!IsOk)
    {
      char *NewCurveSettings = NULL;
      InitializeCurveParams(CurveParams);
      NewCurveSettings = CreateCurveSettingsString(*CurveParams);

      if (NewCurveSettings != NULL)
        {
          TecUtilCurveSetExtendedSettings(XYMapNum,NewCurveSettings);
          TecUtilStringDealloc(&NewCurveSettings);
        }
      if (ErrMsg != NULL)
        {
          TecUtilDialogErrMsg(ErrMsg);
          TecUtilStringDealloc(&ErrMsg);
        }
    }
}
```

Notice at the bottom of this function we repair the **`CurveSettings`** string if it was invalid. It could be that the syntax was wrong, or that the string had not yet been initialized. Either way, we call the function **`InitializeCurveParams()`** in which we setup the **`CurveParams_s`** structure with default values. Then, we create a new **`CurveSettings`** string, which is constructed with the default values. Finally, we set the **`CurveSettings`** string for the current XY-map, **`XYMapNum`**, by calling **`TecUtilCurveSetExtendedSettings()`**.

THE INITIALIZECURVEPARAMS() FUNCTION

Examine the following code in **engine.c**:

```
void InitializeCurveParams(CurveParams_s *CurveParams)
{
  CurveParams->UseIndVarRange = FALSE;
  CurveParams->IndVarMin      = -LARGEDOUBLE;
  CurveParams->IndVarMax      = LARGEDOUBLE;
}
```

Now that we have the laid groundwork for handling the **CurveSettings** string, we can move on to creating the rest of the add-on.

REGISTERING THE ADD-ON WITH TECPLOT

The first thing that must happen when the add-on is loaded into Tecplot is that it must be registered. In **main.c** there is a function:

```
      TecUtilCurveRegisterExtCrvFit(ADDON_NAME,
                                    XYDataPointsCallback,
                                    NULL,  /* ProbeValueCallback */
                                    CurveInfoStringCallback,
                                    CurveSettingsCallback,
                                    AbbreviatedSettingsStringCallback);
```

This function will register the curve-fit add-on with Tecplot. Notice that parameter three is **NULL**. This is because we are not adding the **ProbeValueCallback**.

Notice the version checking code in **main.c** as well. This is required since the extended curve-fit feature is unique to Tecplot Versions 9 and later.

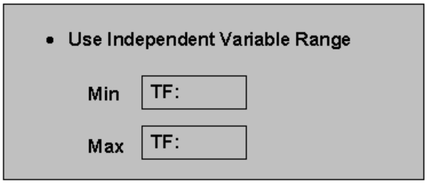
At this point verify that the add-on will compile and load into Tecplot.

CREATING THE DIALOG

In this step we will create the dialog that will be displayed when the user clicks Curve Settings on the Mapping/Zone Style' Curves page when the Curve Type is of type **SimpAvg**. *Please note that we highly recommend that the curve-fit dialog be modal.*

The dialog will have five controls, one toggle, two text fields, and two labels.

1. Load **`gui.lay`** into Tecplot, select Tecplot GUI Builder from the Tools menu and edit the layout as follows:



There will be callbacks associated with each of the text fields, and the toggle button. So that TGB will create meaningful variable names for these controls, we will change their properties in Tecplot. Also, notice that the **`CurveParams_s`** structure has members for each text field and the toggle.

Note: Although the text fields and buttons are referred to as controls, they are in reality Tecplot text field objects, since they exist in a layout file.

2. Double-click on the Use Independent Variable Range toggle and select Options. In the Macro Function field, type **`VarName=UseIndVarRange`**. This will give the callback a meaningful name.

3. Appropriate names for the text fields are IndVarMin and IndVarMax. Although we will not be performing any operations in the text field callbacks, giving them meaningful names is recommended. Set the labelnames to "**`VarName=Min`**" and "**`VarName=Max`**".

4. Now, double-click on the dialog frame and verify that the frame name is as follows:
`ID=1 MODE=MODAL TITLE="Simple Average"`

5. Click Go Build on the TGB dialog.

Now that TGB has created new stub files, be sure to copy the toggle and text field callbacks from **`guicb.tmp`** into **`guicb.c`**.

LAUNCHING AND INITIALIZING THE DIALOG

The add-on dialog is launched by the **`CurveSettingsCallback()`** function in **`engine.c`**. The parameter **`XYMapSet`** is the set of XY-maps that were selected in the Plot-Attributes dialog at the time Curve Settings was clicked. The parameter **`XYMapSettings`** is a string list containing the **`CurveSettings`** strings of all the XY-maps in the set, **`XYMapSet`**.

When Curve Settings is clicked, the function **`CurveSettingsCallback()`** is called by Tecplot. In this function we will save the **`XYMapSet`** and **`XYMapSettings`** so we can use them later in the **`guicb.c`** module. These variables are needed in **`guicb.c`** in order to properly initialize the dialog fields.

In **`engine.c`** verify that **`CurveSettingsCallback()`** is as follows:

```
void STDCALL CurveSettingsCallback(Set_pa        XYMapSet,
                                   StringList_pa XYMapSettings)
{

  TecUtilLockStart(AddOnID);
  /*
   * Save off XYMapSettings and SelectedXYMaps for use
   * in the functions in guicb.c
   */
  GlobalCurve.XYMapSet      = XYMapSet;
  GlobalCurve.XYMapSettings = XYMapSettings;

  /* Build and Launch the dialog */
  BuildDialog1(MAINDIALOGID);
  TecGUIDialogLaunch(Dialog1Manager);

  TecUtilLockFinish(AddOnID);
}
```

GlobalCurve is a global structure that maintains the curve settings when the dialog is launched. This structure must be declared in ENGINE.h as follows:

```
typedef struct
  {
    StringList_pa XYMapSettings;
    Set_pa        XYMapSet;
  } GlobalCurve_s;
```

Now declare the variable **`GlobalCurve`** in **`engine.c`**. Just below the **`#include`** statements in **`engine.c and guicb.c`**, type the following:
`GlobalCurve_s GlobalCurve;`

Finally, make sure the line:
`#include "ENGINE.h"`

exists in **`guicb.c`**.

Initializing the Dialog

Initialization of the dialog is taken care of in **`guicb.c`** in the function **`Dialog1Init_CB()`**. When initializing the dialog, we must place the correct values into each field, and we must also set the sensitivities of each field. In the case of this dialog the sensitivities are as follows:

- **`UseIndVarRange:`** Toggle, always active.
- **`IndVarMin:`** Text field, active when **`UseIndVarRange`** is checked.
- **`IndVarMax:`** Text field, active when **`UseIndVarRange`** is checked.
- **`Min:`** Label, active when **`UseIndVarRange`** is checked.
- **`Max:`** Label, active when **`UseIndVarRange`** is checked.

To set the sensitivities we create the following function in **`guicb.c`**. Be sure this function is placed above the **`Dialog1Init_CB()`** function:

```
static void UpdateMainDialogSensitivities(void)
{
  Boolean_t Sensitive = TecGUIToggleGet(UseIndVarRan_TOG_D1);
  TecGUISetSensitivity(IndVarMin_TF_D1, Sensitive);
  TecGUISetSensitivity(IndVarMax_TF_D1, Sensitive);
  TecGUISetSensitivity(Min_LBL_D1,      Sensitive);
  TecGUISetSensitivity(Max_LBL_D1,      Sensitive);
}
```

If only one XY-map is selected, the **XYMapSettings** string list will have only one member, and that member will be the **CurveSettings** for that mapping. However, when there is more than one mapping selected, and they have different curve settings, how do we decide to initialize the fields on the dialog? Use the following method:

- If all mappings have the same values for any particular field, that value will be used.
- If the selected mappings have different values for any particular field, the default value is used.

To help initialize the fields, we will create a function that will determine the proper value for each variable. The function will then return the appropriate value: the default value if the maps have different settings for that value, or the value that is set if all maps have the same setting for that value. The function is defined below.

The following function function is in **guicb.c**:

```
static void InitializeGUICurveParams(CurveParams_s *CurveParamsPtr)
{
  char          *CurveSettings = NULL;
  CurveParams_s OrigCurveParams;
  Boolean_t    UseIndVarRangeIsSame = TRUE;
  Boolean_t     IndVarMinIsSame = TRUE;
  Boolean_t     IndVarMaxIsSame = TRUE;
  int           ii;
  int           NumMembers;

  /* Get the CurveParams associated with the first mapping. */
  CurveSettings = TecUtilStringListGetString(GlobalCurve.XYMapSettings, 1);
  GetValuesFromCurveSettings(

(EntIndex_t)TecUtilSetGetNextMember(GlobalCurve.XYMapSet,TECUTILSETNOTMEMBER
),
                             CurveSettings,
                             &OrigCurveParams);
  if (CurveSettings != NULL)
    TecUtilStringDealloc(&CurveSettings);

  NumMembers = TecUtilStringListGetCount(GlobalCurve.XYMapSettings);

  /*
   * Compare the value of the first mapping with all the other mappings.
   * This loop will not be done if there is only one mapping selected.
   */
  for (ii = 2; ii <= NumMembers; ii++)
    {
      CurveParams_s TmpParams;
      CurveSettings = TecUtilStringListGetString(GlobalCurve.XYMapSettings,
ii);
      GetValuesFromCurveSettings(

(EntIndex_t)TecUtilSetGetNextMember(GlobalCurve.XYMapSet, ii),
                                 CurveSettings,
                                 &TmpParams);
      if (UseIndVarRangeIsSame)
```

```
          UseIndVarRangeIsSame = (TmpParams.UseIndVarRange ==
                                  OrigCurveParams.UseIndVarRange);

        if (IndVarMinIsSame)
          IndVarMinIsSame = (TmpParams.IndVarMin == OrigCurveParams.IndVarMin);

        if (IndVarMaxIsSame)
          IndVarMaxIsSame = (TmpParams.IndVarMax == OrigCurveParams.IndVarMax);

        if (CurveSettings != NULL)
          TecUtilStringDealloc(&CurveSettings);
      }

  /*
   * Initialize the CurveParamsPtr to the default values.
   * If all mappings have the same value for a particular parameter,
   * use that value instead.
   */
  InitializeCurveParams(CurveParamsPtr);

  if (UseIndVarRangeIsSame)
    CurveParamsPtr->UseIndVarRange = OrigCurveParams.UseIndVarRange;
  if (IndVarMinIsSame)
    CurveParamsPtr->IndVarMin = OrigCurveParams.IndVarMin;
  if (IndVarMaxIsSame)
  CurveParamsPtr->IndVarMax = OrigCurveParams.IndVarMax;
}
```

Finally we will add the following function. This function will initialize the dialog fields and will be called from the **Dialog1Init_CB()** function as described below. This function also calls InitializeGUICurveParams() which was previously defined. The **TecGUIextFieldSetDouble** () functions are convenience functions defined in the adkutil.c module.

To use these functions, be sure to add the following line to the top of **guicb.c**:
#include "ADKUTIL.h"

The following function is in **guicb.c** below the **UpdateMainDialogSensitivities()** and below the **InitializeGUICurveParams()** function:

```
static void UpdateMainDialog(void)
{
  CurveParams_s CurveParams;
  InitializeGUICurveParams(&CurveParams);
  TecGUIToggleSet(UseIndVarRan_TOG_D1,CurveParams.UseIndVarRange);
  TecGUITextFieldSetDouble(IndVarMin_TF_D1,CurveParams.IndVarMin,"%G");
  TecGUITextFieldSetDouble(IndVarMax_TF_D1,CurveParams.IndVarMax,"%G");
  UpdateMainDialogSensitivities();
}
```

At this point it is recommended that you compile and run your add-on to make sure that the fields and sensitivities are initialized correctly. The dialog should appear with the Use Independent Variable Range toggle off, and the remaining controls should be insensitive. Using the Use Independent Variable Range toggle will not change the sensitivities of the dialog at this point.

MAKING THE DIALOG OPERATIONAL

To make the dialog fully operational, there are two things that must be done. The first is to update the sensitivities of the text field controls when **Use Independent Variable Range** toggle is clicked. The second is to make the dialog set the values when **OK** is clicked.

Updating the Sensitivities

To be sure that the text field sensitivities are updated when the toggle button is pressed as follows:

```
static void UseIndVarRan_TOG_D1_CB(const int *I)
{
  TecUtilLockStart(AddOnID);
  /* Make sure to update the sensitivities when the toggle button is pressed.
*/
  UpdateMainDialogSensitivities();
  TecUtilLockFinish(AddOnID);
}
```

The process to follow when **OK** is clicked is:

1. Collect the information from the dialog.

2. Create a new `CurveSettings` string.

3. Call `TecUtilXYMapSetCurve()` with the appropriate parameters to set the extended curve settings for the set of XY-maps.

4. Drop the dialog.

The following function collects the information from the dialog and places it into the `CurveParams` structure. The function will use the `TecGUIextFieldGetDouble()` function, which is defined in the `adkutil.c` module.

The following function is in `guicb.c` above the `Dialog1OkButton_CB()` function:

```
static void AssignCurveParams(CurveParams_s *CurveParams)
{
  CurveParams->UseIndVarRange = TecGUIToggleGet(UseIndVarRan_TOG_D1);
  /*
   * Note this function returns a boolean alerting user whether or not
   * input value is legitimate.  Some error checking may be added here.
   */
  TecGUITextFieldGetDouble(IndVarMin_TF_D1,&CurveParams->IndVarMin);
  TecGUITextFieldGetDouble(IndVarMax_TF_D1,&CurveParams->IndVarMax);
}
```

The `Dialog1OkButton_CB()` function to look as follows:

```
static void Dialog1OkButton_CB(void)
{
  /* Only unlock tecplot here because a modal dialog was launched. */

  /* When curve settings change, Tecplot must be informed of the change. */

  char *CurveSettings = NULL;
  CurveParams_s CurveParams;

  /* Assign the new curve parameters from the dialog settings. */
  AssignCurveParams(&CurveParams);

  /* Create the Curve Settings string from the new curve parameters. */
  CurveSettings = CreateCurveSettingsString(CurveParams);
  if (CurveSettings != NULL)
    {
      EntIndex_t  Map;
      TecUtilSetForEachMember(Map, GlobalCurve.XYMapSet)
        {
          TecUtilCurveSetExtendedSettings(Map, CurveSettings);
        }
      TecUtilStringDealloc(&CurveSettings);
    }

  TecGUIDialogDrop(Dialog1Manager);
  TecUtilLockFinish(AddOnID);
}
```

At this point, the dialog should be fully functional. The dialog will be initialized with the correct values and sensitivities. The sensitivities will be updated correctly, and Tecplot will be informed when the **CurveSettings** string is changed.

UPDATING THE MAPPING/ZONE STYLE DIALOG

To update the Mapping/Zone Style dialog, we move back to the **engine.c** module. The **CurveSettings** field of the Mapping/Zone Style dialog will be filled with the string returned by the **AbbreviatedSettingsStringCallback()** function. If this function is undefined, or returns a value of **NULL**, the **CurveSettings** string that Tecplot stores will be used in the Mapping/Zone Style dialog.

To create this string, we will evaluate the **CurveSettings** string, and create a legible output string. The string we will produce will look like:

- If using the Independent Variable Range, IndVarMin = 2 and IndVarMax = 7:
  "IndVarRange: Min = 2; Max = 7"
- If not using the Independent Variable Range:
  "No IndVarRange"

```
void STDCALL AbbreviatedSettingsStringCallback(EntIndex_t XYMapNum,
                                              char      *CurveSettings,
                                              char     **AbbreviatedSettings)
{
  CurveParams_s CurveParams;
  char          *S;

  TecUtilLockStart(AddOnID);
  GetValuesFromCurveSettings(XYMapNum,
                             CurveSettings,
                             &CurveParams);

  S = TecUtilStringAlloc(80, "Abbreviated Settings");

  if (CurveParams.UseIndVarRange)
    {
      sprintf(S,
              "IndVar Range: Min = %G; Max = %G",
              CurveParams.IndVarMin,
              CurveParams.IndVarMax);
      *AbbreviatedSettings = S;
    }
  else
    {
      strcpy(S, "No IndVarRange");
      *AbbreviatedSettings = S;
    }
  TecUtilLockFinish(AddOnID);
}
```

At this point, it is recommended that you compile the add-on and verify that you can change the settings via your dialog, and that settings are displayed on the Mapping Style dialog.

THE CURVE-FIT

The curve-fit is almost complete given the code created by the **CreateNewAddOn** script or the Tecplot Add-on Wizard. The curve-fit computes the average of the data. We alter the curve-fit to exclude points that fall outside the range specified in the dialog.

THE XYDATAPOINTSCALLBACK()

We will need to alter the **XYDataPointsCallback()** to determine the proper independent variable range. This range is the range limited by the extents of the data and the values specified in the Curve-Fit dialog. Alter the **XYDataPointsCallback()** as follows:

```
Boolean_t STDCALL XYDataPointsCallback(FieldData_pa RawIndV,
                                       FieldData_pa RawDepV,
                                       CoordScale_e IndVCoordScale,
                                       CoordScale_e DepVCoordScale,
                                       LgIndex_t    NumRawPts,
                                       LgIndex_t    NumCurvePts,
                                       EntIndex_t   XYMapNum,
                                       char        *CurveSettings,
                                       double      *IndCurveValues,
                                       double      *DepCurveValues)
{
```

```
Boolean_t IsOk  = TRUE;

int       ii;
double    Average;
double    Delta = 0.0;
double    IndVarMin,
IndVarMax;
CurveParams_s CurveParams;

TecUtilLockStart(AddOnID);

/* Get the min and max values of the independent variable. */
TecUtilDataValueGetMinMaxByRef(RawIndV,
                               &IndVarMin,
                               &IndVarMax);

/* Get the curve parameters */
GetValuesFromCurveSettings(XYMapNum,
                           CurveSettings,
                           &CurveParams);

if (CurveParams.UseIndVarRange)
  {
    /*
     * Adjust the independent variable range to fall either within
     * the range of data or the range specified by the
     * CurveParams structure.
     */
    IndVarMin = MAX(IndVarMin, CurveParams.IndVarMin);
    IndVarMax = MIN(IndVarMax, CurveParams.IndVarMax);
  }

Delta = (IndVarMax-IndVarMin)/(NumCurvePts-1);

/*
 * Find the average value of the raw dependent variable for the
 * default curve fir (straight line at average).
 */
Average = SimpleAverage(RawDepV,
                        RawIndV,
                        NumRawPts,
                        IndVarMin,
                        IndVarMax);

/*
 * Step through all the points along the curve and set the
 * DepCurveValues to the Average at each IntCurveValue.
 */

for (ii = 0; ii < NumCurvePts; ii++)
  {
    IndCurveValues[ii] = ii*Delta + IndVarMin;
```

```
      DepCurveValues[ii] = Average;
    }

  TecUtilLockFinish(AddOnID);
  return IsOk;
}
```

Notice that the **SimpleAverage()** function has also been changed. We are now passing more information to the **SimpleAverage()** function so it can make the decision about what points to include in the average value calculation. Alter the **SimpleAverage()** function as follows:

```
/**
 * Function to compute the average of the raw dependent variable for the
 * default fit (straight line at average).
 *
 * REMOVE THIS FUNCTION FOR OTHER FITS.
 */
double SimpleAverage(FieldData_pa RawDepV,
                     FieldData_pa RawIndV,
                     LgIndex_t    NumRawPts,
                     double       IndVarMin,
                     double       IndVarMax)
{
  int    ii;
  int    Count   = 0;
  double Sum     = 0;

  for (ii = 0; ii < NumRawPts; ii++)
    {
      double IndV = TecUtilDataValueGetByRef(RawIndV, ii+1);

      /*
       * Only compute the average on values that fall in the
       * specified range of the independent variable.
       */
      if ( IndV >= IndVarMin && IndV <= IndVarMax)
        {
          Sum += TecUtilDataValueGetByRef(RawDepV, ii+1);
          Count++;
        }
    }

  return (Sum/Count);
}
```

The **SimpleAverage()** function is also used in the **CurveInfoStringCallback()** so we will have to alter that function as well. You will notice that the process in **CurveInfoStringCallback()** is very similar to the process used in **XYDataPointsCallback()**. The **CurveInfoStringCallback()** function looks as follows:

```
Boolean_t STDCALL CurveInfoStringCallback(FieldData_pa RawIndV,
                                          FieldData_pa RawDepV,
                                          CoordScale_e IndVCoordScale,
                                          CoordScale_e DepVCoordScale,
                                          LgIndex_t    NumRawPts,
                                          EntIndex_t   XYMapNum,
                                          char        *CurveSettings,
                                          char       **CurveInfoString)
{
  Boolean_t     IsOk = TRUE;
  CurveParams_s CurveParams;
  double IndVarMin,IndVarMax;
  double Average;

  TecUtilLockStart(AddOnID);

  /*
   * If this function is not registered with Tecplot, no curve
   * information will be displayed in the XY-Curve Info dialog.
   */
  *CurveInfoString = TecUtilStringAlloc(30, “CurveInfoString”);

  /* Get the curve parameters. */
  GetValuesFromCurveSettings(XYMapNum,CurveSettings,&CurveParams);

  if (CurveParams.UseIndVarRange)
    {
      /*
       * Adjust the Independent variable range to fall either within
       * the range of the data or the range specified by the
       * CurveParams structure.
       */
      IndVarMin = CurveParams.IndVarMin;  /* initialize these values */
      IndVarMax = CurveParams.IndVarMax;
      IndVarMin = MAX(IndVarMin, CurveParams.IndVarMin);
      IndVarMax = MIN(IndVarMax, CurveParams.IndVarMax);
    }

  Average = SimpleAverage(RawDepV,
                          RawIndV,
                          NumRawPts,
                          IndVarMin,
                          IndVarMax);

  sprintf(*CurveInfoString, “Average is: %G\n”, Average);

  TecUtilLockFinish(AddOnID);
  return IsOk;
}
```

The add-on is now complete. You should compile the add-on at this time and verify that it works as expected.

As a further exercise, add error-checking to the dialog so that the minimum value is greater than the maximum value.

The process described in this manual is the preferred process for creating curve-fit add-ons with configurable settings. Whenever creating an add-on of this type, you should refer to this example as a template.

INDEX