2 Programming the User Interface

- How UPI Works 4-12
- UPI Include Files 4-20
- Running an Interpreted Macro 4-21
- Running a Compiled Macro 4-22
- Creating an Interpreted Macro 4-23
- Creating a Compiled Macro (DLL) 4-27
- Binding Macros to Hot Keys 4-36
- Binding Macros to Menu Items 4-38
- Debugging Interpreted Macros 4-39

(continued)

- Debugging Compiled Macros 4-40
- Creating a Layout Palette 4-41
- Copy-Protecting Macro DLLs 4-50

How UPI Works

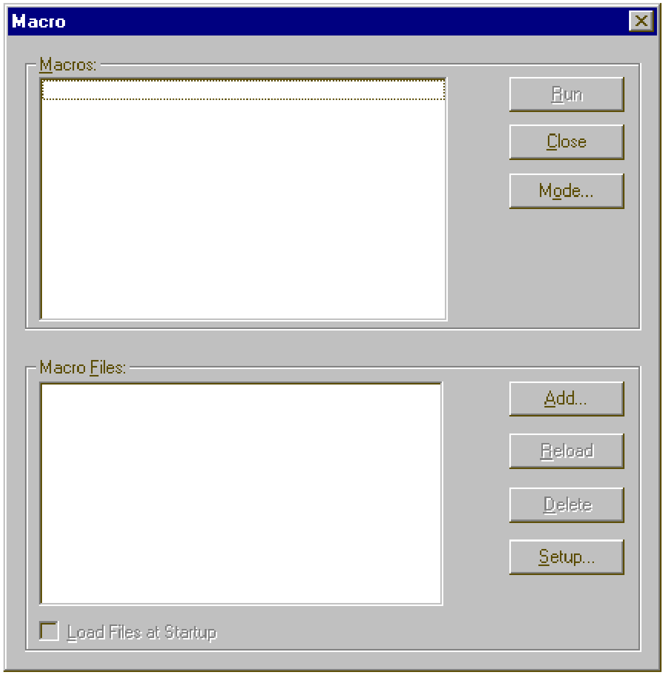
Macros are written using UPI function calls, then loaded and run via the macro interface. The macro interface also allows you to specify the UPI operating modes and interpreter setup options.

For more information on the UPI function calls, see the UPI Functions Reference on page 4-55.

Macro Interface

The **Macro** dialog is the interface to the UPI macros. It allows you to create, edit, and run macros, and set the UPI operating mode.

Choose **Tools > Macro** to open the **Macro** dialog.



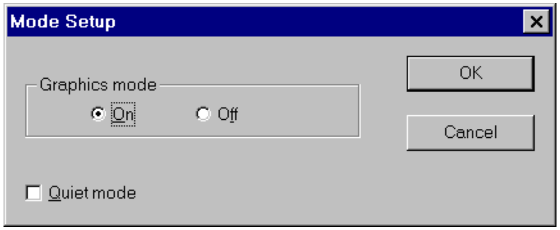
Options include:

Macros	The list of registered macros. Click on a macro to select it.
Macro Files	The name and complete paths of all loaded macro files. Click on a macro file to select it.
Load Files at Startup	When this box is checked, the **Macro Files** listed are loaded when the application is started.
Run	Executes the macro highlighted in the **Macros** list.
Close	Closes the **Macro** dialog.
Mode	Opens the **Mode Setup** dialog where you specify the UPI operating mode.
Add	Invokes the **Open** dialog to load a macro file and add it to the **Macro Files** list.
Reload	Reloads the selected macro.
Delete	Unregisters all macros defined in the highlighted macro file and unloads the file from memory.

Setup	Opens the **Setup Application—UPI** dialog for specifying the interpreter setup parameters. See UPI on page 1-123 for further information.

UPI Operating Modes

Click **Mode** in the **Macro** dialog to specify the UPI operating mode. The **Mode Setup** dialog will appear.



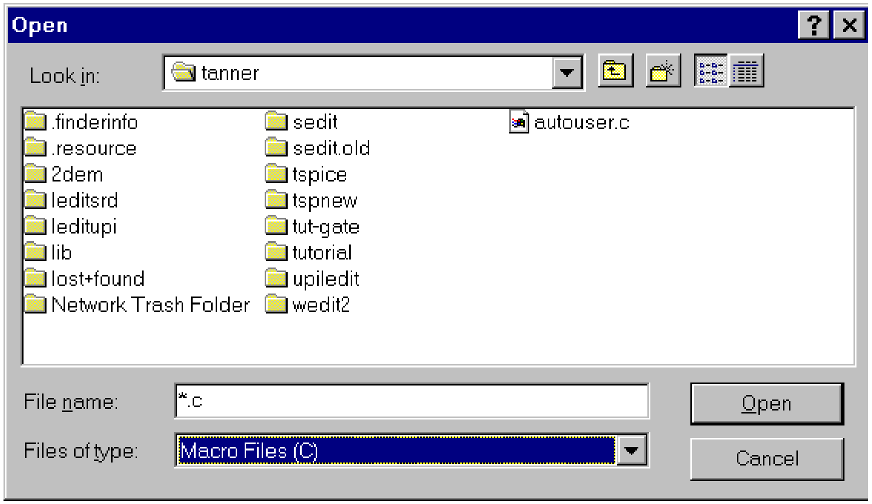
Graphics mode: [✓ On] Off

☐ Quiet mode

Graphics mode	When **On**, screen is updated after every relevant UPI call in a user-defined macro.When **Off**, screen is refreshed only on completion of the macro.
Quiet mode	Suppresses alert boxes. The use of quiet mode is required for batch processing.

Adding a Macro

Clicking **Add** in the **Macro** dialog calls the **Open** dialog.



File name: *.c

Files of type: Macro Files (C)

You can add a macro in one of two ways:

- As C code which must be interpreted
- As a compiled, dynamic-link library (DLL).

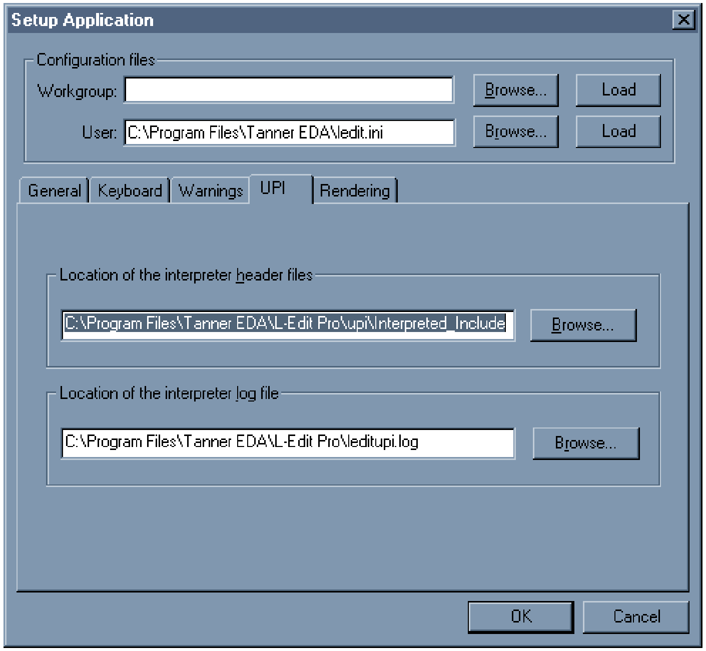
The only difference between a C macro file and a DLL is in the speed of loading and execution. A C file must be interpreted before its macros will appear in the **Macros** list of the **Macro** dialog. L-Edit has a bult-in C interpreter.

A DLL, however, is directly loaded into memory and all the macros appear in the **Macros** list immediately. You can create a C macro with any text editor. To create a DLL macro, however,you must use a C/C++ compiler such as Microsoft Developer Studio.

Interpreter Setup

Before running an interpreted macro, you must first set a path to the header files that L-Edit uses to interpret a macro. You must also set a path to the log file where UPI writes macro errors.

To perform these tasks, click **Setup** in the **Macro** dialog. Alternately, you can choose **Setup > Application**. When L-Edit displays the **Setup Application** dialog, click the **UPI** tab:


Setup Application

Configuration files

Workgroup:

User: C:\Program Files\Tanner EDA\ledit.ini

Location of the interpreter header files

C:\Program Files\Tanner EDA\L-Edit Pro\upi\Interpreted_Include

Location of the interpreter log file

C:\Program Files\Tanner EDA\L-Edit Pro\leditupi.log

Options include:

Location of the interpreter header files	The complete path of the directory containing the L-Edit interpreter header files. Clicking **Browse** next to this field calls a standard Windows directory browser.
Location of the interpreter log file	The name of the log file to which macro errors will be written. Clicking **Browse** next to this field calls a standard Windows directory browser. Be sure you have write permission to the directory where you create your log file. Log files are only used when running interpreted macros.

UPI Include Files

The directory **L-Edit Pro\upi\Interpreted_Include** contains a number of files required by the L-Edit C interpreter. The L-Edit C interpreter only supports functions defined in these header files.

The file **ldata.h** is a standard UPI include file that contains function prototypes of all UPI functions, including definition of the **Interface Functions** (page 4-57). The file **lupi_usr.h** also contains definitions of all the supported hot key combinations.

The files **ctype.h**, **malloc.h**, **math.h**, **stdarg.h**, **stdio.h**, **stdlib.h**, **string.h**, and **time.h** are L-Edit versions of standard C language header files.

The files **upistub.c** and **upistub.h** are required by the L-Edit C interpreter.

Running an Interpreted Macro

The following procedure explains how to run an interpreted macro.

☑ Start L-Edit.

☑ Use **File > Open** to open **L-Edit Pro\Samples\Upi\upisampl.tdb**.

☑ Select **Tools > Macro** to call the **Macro** dialog.

☑ Click **Setup** to specify interpreter parameters (the locations of the header and log files).

☑ Click **Add** and select a macro file with a **.c** extension. For example, load the file **L-Edit Pro\Samples\Upi\intrpted\mosfet\mosfet.c**.

☑ Select the macro from the **Macros** list and click **Run**, or double click on the macro name in the dialog to execute it.

Running a Compiled Macro

This procedure explains how to run a compiled macro.

☑ Start L-Edit.

☑ Use **File > Open** to open **L-Edit Pro\Samples\Upi\upisampl.tdb**.

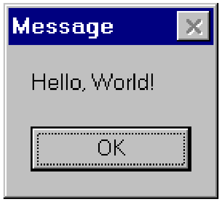
☑ Select **Tools > Macro** to call the **Macro** dialog.

☑ Click **Add** and select a macro file with a **.dll** extension. For example, load the file **L-Edit Pro\Samples\UPI\dll\resistor\release\resistor.dll**.

☑ Select the macro from the **Macros** list and click **Run**, or double click on the macro name in the dialog to execute it.

Creating an Interpreted Macro

This lesson will show you how to build an interpreted UPI application. The sample application displays a message box like the one shown here:



There are four steps to creating this application:

☑ Create a module outline.

☑ Write a function to display the message box.

☑ Register this function as a macro.

☑ Save the code as a **.c** file.

In outline form, your code will look like this:

```
module <modulename> {

     <include files>

     <macro function 1 >
     <macro function 2 >
     <macro function 3 >
          .
          .
          .

     <macro registration function>

}
{call to macro registration function}
```

Now, add code to display the message box:

```
void HelloMacro( void )
{
    LDialog_MsgBox ( "Hello, World!" );
}
```

LDialog_MsgBox is a UPI call that displays a message box. When executed, this function will show the string "Hello, World!" in a message box.

Now write a function that will register the **Hello World** macro function.

```
void hello_world_macro_register ( void )
{
    LMacro_Register ( “Hello, World!”, “HelloMacro” );
}
```

The function **hello_world_macro_register** registers the function **HelloMacro** as an available macro. The function **LMacro_Register()** associates the name of the macro as presented in the **Macro** dialog (Hello, World!) with the function **HelloMacro**. Note that the second parameter—**HelloMacro**—is the name of the function to be called, in quotes.

If multiple user macro functions are defined, the **hello_world_macro_register** function should register each of them individually.

The complete code for the **Hello World** macro is as follows:

```
module Hello_World_module {

#include “ldata.h”

void HelloMacro( )
{
    LDialog_MsgBox ( “Hello, World!” );
}

void hello_world_macro_register ( void )
{
    LMacro_Register ( “Hello, World!”, “HelloMacro” );
}

}
hello_world_macro_register();
```

The first line contains the module name. The module name should be unique so that it will not replace another module loaded at the same time. In this case, the module name is **Hello_World_module**.

Save your code using the **.c** filename extension. Use this filename whenever you name a macro file that you want L-Edit to recognize as an interpreted macro.

Creating a Compiled Macro (DLL)

In this lesson, you will learn how to write a UPI macro that will generate a simple layout at the press of a key. This macro draws an Active-to-Metal contact at the current mouse location. You will also learn how to bind a macro to a hot key.

Creating a compiled macro is similar to creating an interpreted one, with one additional step—compiling the DLL. That makes a total of five steps to create a DLL:

☑ Create a DLL outline.

☑ Write a function to draw the Active-to-Metal contact.

☑ Register this function as a macro in the **UPI_Entry_Point** function.

☑ Save the code as a **.c** file.

☑ Compile the DLL.

In outline, the code for your DLL will look like this:

```
<include files>

<macro function 1 >
<macro function 2 >
```

```
<macro function 3 >
    .
    .
    .

<UPI_Entry_Point function>
```

Now add code to draw an Active-to-Metal contact at the current mouse location.

```
void Contact_Active_Metal1_Macro ( )
{
    LCell  Cell_New    = LCell_GetVisible ( );
    LFile  File_New    = LCell_GetFile ( Cell_New );
    LLayer Layer_Active = LLayer_Find ( File_New, “Active”
   );
    LLayer Layer_Metal1 = LLayer_Find ( File_New, “Metal1”
   );
    LLayer Layer_ActCnt = LLayer_Find ( File_New,
   “ActivContact” );
    LLayer Layer_N_Sel = LLayer_Find ( File_New, “N Select”
   );
    LPoint Point_Cursor = LCursor_GetPosition ( );

LCoord X, Y;

    X = Point_Cursor.x;
    Y = Point_Cursor.y;
    LBox_New ( Cell_New, Layer_ActCnt, -1 + X, -1 + Y, 1 +
   X, 1 + Y );
```

```
    Box_New ( Cell_New, Layer_Metal1, -2 + X, -2 + Y, 2 + X,
   2 + Y );
    Box_New ( Cell_New, Layer_Active, -3 + X, -3 + Y, 3 + X,
   3 + Y );
    LBox_New ( Cell_New, Layer_N_Sel , -5 + X, -5 + Y, 5 +
   X,5 + Y );
}
```

The function **Contact_Active_Metal1_Macro( )** finds the current cell. It then draws the necessary boxes at the current mouse location to create a contact.

Now write the **UPI_Entry_Point** function. This function is the entry point for the user DLL. You use it to register the macros. In this example, the function will register the **Contact_Active_Metal1_Macro** function.

```
int UPI_Entry_Point( void )
{
    LMacro_BindToHotKey ( KEY_F1, “My Contact, Active-
   Metal1”, “Contact_Active_Metal1_Macro” );
    return 1;
}
```

This function registers the function **Contact_Active_Metal1_Macro** as an available macro. The function **LMacro_BindToHotKey( )** registers the macro and binds it to a specified hot key. Note that the second parameter—**Contact_Active_Metal1_Macro**—is the name of the function to be called, in quotes. If this function returns zero, L-Edit will unload the DLL.

The complete code for the **Active-to-Metal** macro is as follows:

```
#include “ldata.h”
void Contact_Active_Metal1_Macro ( )
{
    LCell  Cell_New   = LCell_GetVisible ( );
    LFile  File_New   = LCell_GetFile ( Cell_New );
    LLayer Layer_Active = LLayer_Find ( File_New, “Active”
   );
    LLayer Layer_Metal1 = LLayer_Find ( File_New, “Metal1”
   );
    LLayer Layer_ActCnt = LLayer_Find ( File_New,
   “ActiveContact” );
    LLayer Layer_N_Sel = LLayer_Find ( File_New, “N Select”
   );
    LPoint Point_Cursor = LCursor_GetPosition ( );
    LCoord X, Y;

    X = Point_Cursor.x;
    Y = Point_Cursor.y;
    LBox_New ( Cell_New, Layer_ActCnt, -1 + X, -1 + Y, 1 +
   X, 1 + Y );
    Box_New ( Cell_New, Layer_Metal1, -2 + X, -2 + Y, 2 + X,
   2 + Y );
    Box_New ( Cell_New, Layer_Active, -3 + X, -3 + Y, 3 + X,
   3 + Y );
    LBox_New ( Cell_New, Layer_N_Sel , -5 + X, -5 + Y, 5 +
   X,5 + Y );
}
```

```
int UPI_Entry_Point( void )
{
    LMacro_BindToHotKey ( KEY_F1, “My Contact,
    Active-Metal1”, “Contact_Active_Metal1_Macro” );
    return 1;
}
```

Save this code as **contact.c** and compile it as **contact.dll**.

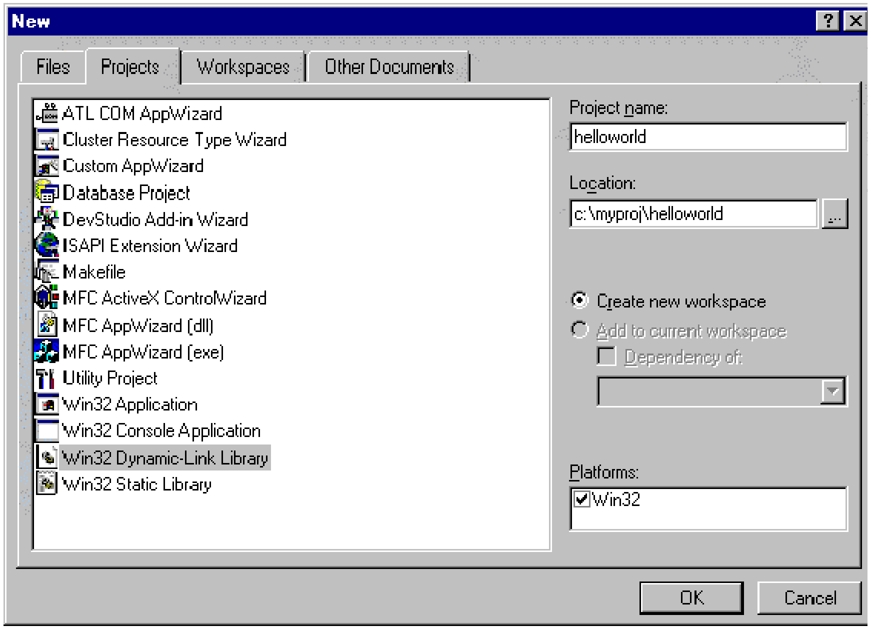
Compiling the DLL

This section describes how to compile your C-language macros as DLLs. The lesson includes an actual example of compiling a DLL.

To compile a DLL, you will need a C/C++ compiler such as Microsoft Visual C++ installed on your system. These instructions are for Microsoft Visual C++ version 6.0. If you have not already installed it, do so now.

☑ Start Microsoft Visual C++.

☑ Select **File > New**. Click the **Projects** tab and set the options shown below, then click **OK**:



ATL COM AppWizard
Cluster Resource Type Wizard
Custom AppWizard
Database Project
DevStudio Add-in Wizard
ISAPI Extension Wizard
Makefile
MFC ActiveX ControlWizard
MFC AppWizard (dll)
MFC AppWizard (exe)
Utility Project
Win32 Application
Win32 Console Application
Win32 Dynamic-Link Library
Win32 Static Library

Project name: helloworld

Location: c:\myproj\helloworld

[● Create new workspace] ○ Add to current workspace

☐ Dependency of:

Platforms: [✓ Win32]

- Select **Win32 Dynamic Link Library** as the project type.

- Type **helloworld** as the **Project name**. The subdirectory **c:\myproj\helloworld** will be created. Use the file browser to navigate to this directory if it already exists.

☑ In the **Win32 Dynamic-Link Library** wizard, select **An empty DLL project**.

☑ Right-click **Source Files** in the **Projects** tree and select **File > New**. In the resulting menu, select **C++ Source File** and click **OK**.

☑ Open the file **L-Edit Pro\Samples\Upi\dll\hworld\hello.c**. Copy the text in this file and paste it into the editing space. Save the file with the name **hello.c**.

```
#include “ldata.h”

void HelloMacro (void)
{
    LDialog_MsgBox(“Hello World”);
}

int UPI_Entry_Point (void)
{
    LMacro_Register(“Hello World”, HelloMacro);
    return 1;
}
```

☑ Next, create a definition file. Right-click **Source Files** in the **Projects** tree again and select **File > New**. In the resulting menu, select **Text File** and click **OK**.

☑ Open the file **L-Edit Pro\Samples\Upi\dll\hworld\hello.def**. Copy the text this file and paste it into the editing space. Save the file with the name **hello.def**.

```
EXPORTS
    UPI_Entry_Point
    HelloMacro
```

☑ Select **Tools > Options** and perform the following actions:

- Click the **Directories** tab. Click the drop-down menu **Show directories for** and select **Include files**. In the **Directories** list, select **L-Edit Pro\upi\DLL_include**.
- Click the drop-down menu **Show directories for** and select **Library files**. In the **Libraries** list, select **L-Edit Pro\upi\DLL_include** to access **upilink.lib**.
- Click **OK**.

☑ Select **Project > Settings** and perform the following actions:

- Click the **Debug** tab. In the field **Executable for debug session**, enter **C:\Tanner\LEdit83\ledit.exe**.
- Click the **C/C++** tab. From the **Category** drop-down menu, select **Code Generation**. Set processor to **80486**. Set **Calling convention** to **__stdcall**. Set **Struct member alignment** to **8 bytes**.
- From the **Category** drop-down menu, select **General**. In the **Settings for** list, select **Debug** and **Release** one at a time. For each one, add **/Tp** at the

end of **Project Options**. This instructs the compiler to run **.c** files through the C++ compiler.

- From the **Category** drop-down menu, select **Preprocessor**. Add **MAKE_DLL** to the **Preprocessor definitions** list**.** Set **Setting for** to both the Win32 Debug and Win32 Release versions.
- Select **All Configurations**. Click the **Link** tab. Make sure that **Object/library modules** has **upilink.lib** at the end for both the Win32 Debug and Win32 release versions.
- Click **OK**.

☑ Set **Active Configuration** and choose **Debug** version.

☑ Select **Build > Build** *your_DLL* to compile your DLL.

You have just learned how to create and compile a macro DLL. In the next lesson, you will learn how to bind your macro to a hot key.

Binding Macros to Hot Keys

You can modify the L-Edit user interface so that you can execute your macro from a hot key or a menu item as well as from the **Macro** dialog. The process is known as *binding* the macro to a hot key or menu item.

You bind a macro to a hot key using **LMacro_BindToHotKey** (page 4-86). This function registers the macro and binds it to the specified hot key.

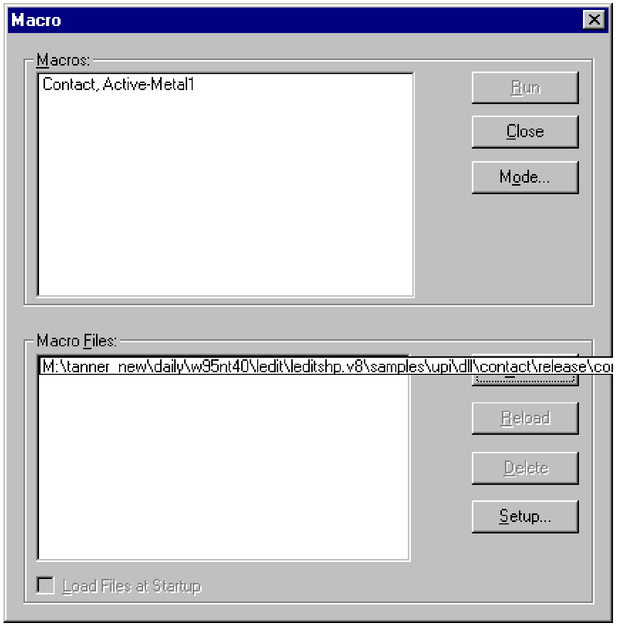
For example, the code:

```
LMacro_BindToHotKey ( KEY_F1, “Contact, Active-Metal1”,
    “Contact_Active_Metal1” );
```

binds the macro **Contact, Active-Metal1** to the **F1** key.

All the allowed key codes are defined in **ldata.h**.

When a function contains this call, it will appear in the **Macro** dialog in this way:



Contact, Active-Metal1

M:\tanner_new\daily\w95nt40\ledit\leditshp.v8\samples\upi\dll\contact\release\co

In this example, the macro **Contact, Active-Metal1** is bound to the **F1** key. The macro thus overwrites the previous **F1** key binding, if any exists. To view key bindings for all macros, use **Keyboard Customization** (page 1-119).

Binding Macros to Menu Items

This section describes how to make your macro accessible as an L-Edit menu command. This process is called *binding* the macro to the menu item.

The UPI call **LMacro_BindToMenu** (page 4-87) will register the macro and bind it to a user-specified menu item.

```
void LMacro_BindToMenu( char *menu, char *macro_desc, void
    *function);
```

For example, the following command will bind **Contact_Active_Metal1** to the **Tools** menu.

```
LMacro_BindToMenu ( “Tools”, “My Contact, Active-Metal1”,
    “Contact_Active_Metal1” );
```

Debugging Interpreted Macros

Here are some useful hints for debugging an interpreted macro:

- Make sure that the interpreter header file location is properly set. You set the header file location using the **Setup Application** dialog—see Interpreter Setup on page 4-17.
- Make sure the C-library functions in your code are present in the header files specified in the **Interpreter Setup** dialog. If they are not present, L-Edit will not be able to interpret your code.
- Make sure you have proper permissions to the directory where you create your log file.

Debugging Compiled Macros

Here are some useful hints for debugging a compiled macro:

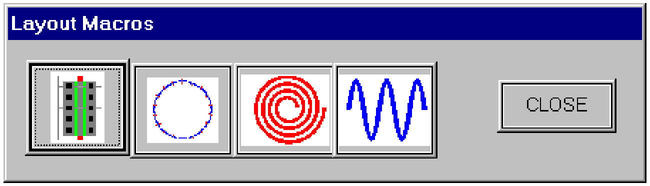
- Compile the DLL with debugging symbols.
- Set a break point in **UPI_Entry_Point()** function in the DLL.

When you execute the macro, the debugger will stop at **UPI_Entry_Point()** function.

Although you cannot step through the UPI functions, you can see the parameters that are passed and the entire control flow of the macro DLL.

Creating a Layout Palette

In this section, you will learn to create a layout palette that always stays on screen. To follow this lesson, you will need some knowledge of the Windows application programming interface (API).



Each button on this layout palette represents a user macro that creates layout geometry such as a MOSFET, spiral, or gear.

The layout palette is called a modeless dialog because it can remain open while you perform other work in L-Edit. You can use the Windows API to associate tool tips and bitmaps with its buttons.

To implement a layout palette, you must write a DLL with a **UPI_Entry_Point()** function that registers the layout palette macro. Running that macro will make the layout palette appear on your screen.

The following procedure explains how to create a DLL that brings up a layout palette.

- ☑ Use a resource editor to create resources for the layout palette.
- ☑ Create a **UPI_Entry_Point()** function that registers the layout palette macro.
- ☑ Write code for displaying and managing the layout palette.
- ☑ Write macros that will generate layout for every button in your layout palette—for example, **gear.c**, **mosfet.c**, **polarary.c**, **spiral.c**, or **spring.c**.
- ☑ Compile the DLL. To review the procedure for compiling a DLL, see Creating a Compiled Macro (DLL) on page 4-27.

Creating Resources

Use a resource editor such as the one provided with Visual C++ to create resources for the layout palette. You will need to create the modeless dialog, push buttons, and bitmaps.

Give this file a name such as **dll.rc**.

The following sample code contains the **UPI_Entry_Point()** function that registers the layout palette macro. When executed, the layout palette macro will call **MainFunction()**, which displays the layout palette.

You can copy this code and use it to create your own layout palette. Save it as **user.c**.

```
#define STRICT
#include “windows.h”
#include “ldata.h”

extern void  LWindow_GetParameters(void **hInst, void
   **hWnd, void **hLib);
extern void MainFunction(HINSTANCE hInst, HWND hWnd,
   HINSTANCE hLib);

HINSTANCE hInst=NULL;
HWND hWnd=NULL;
HINSTANCE hLib=NULL;

void LayoutPalette ( void )
{
    MainFunction(hInst, hWnd, hLib);
}

int UPI_Entry_Point ( void )
{
    LWindow_GetParameters( (void**)&hInst, (void**)&hWnd,
   (void**)&hLib);
    LMacro_Register ( “Layout Palette”, “LayoutPalette” );
    return 1;
}
```

Here is sample code for a file called **dll.c**. This code implements MainFunction, which displays and manages the layout palette.

You can copy this code and use it to create your own layout palette.

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT message, WPARAM
   wParam, LPARAM lParam )
{
switch (message)
{
HANDLE_DLG_MSG( hDlg , WM_COMMAND , DlgOnCommand ) ;
HANDLE_DLG_MSG( hDlg , WM_INITDIALOG , OnInitDialog ) ;
}
return FALSE ;
}


void DlgOnCommand ( HWND hDlg , int iID , HWND hwndCtl ,
   UINT uCodeNotify )
{
 switch( iID )
 {
        case IDC_BUTTON_5:
        MosfetMacro();
        break;
 }
}
```

```
BOOL OnInitDialog ( HWND hDlg , HWND hwndFocus , long
   lInitParam )
{

    HBITMAP hBmp;
    TOOLINFO info;
    HWND hToolTip;

    hBmp = LoadBitmap(l_hLib, MAKEINTRESOURCE(IDB_BITMAP5));
    SendDlgItemMessage(hDlg, IDC_BUTTON_5, BM_SETIMAGE, 0,
   (LPARAM)hBmp);
    FreeResource(hBmp);
return TRUE ;
}

void OnDestroy ( HWND hDlg )
{
    DWORD error;
    if ( hDlg )
        DestroyWindow( hDlg );
    l_hDlg = NULL;
}

void OnClose ( HWND hDlg )
{
    DWORD error;
    DestroyWindow( hDlg );
    l_hDlg = NULL;
}
```

```
void MainFunction(HINSTANCE hInst, HWND hWnd, HINSTANCE
   hLib)
{

    /* Check if the resources have already been loaded */
    if (!loaded) {
        HRSRC hRes;

        hRes = FindResource(hLib, “PaletteDialogBox”,
   RT_DIALOG);
        hResLoad = (HRSRC )LoadResource(hLib, hRes);
        hTmpl = (DLGTEMPLATE *)LockResource(hResLoad);
        hDlg = CreateDialogIndirect(Null, hTmpl, NULL,
   DlgProc);
        l_hDlg = hDlg;
        UnlockResource(hResLoad);
        FreeResource(hRes);
    }
    loaded = 1;
    hDlg = l_hDlg;
    ShowWindow(hDlg, SW_NORMAL);

}

BOOL WINAPI DllMain( HANDLE hDLL, DWORD dwReason, LPVOID
   lpReserved )
{
    switch( dwReason ) {
```

```
    case DLL_PROCESS_DETACH:
        if ( l_hDlg != NULL ) {
            DestroyWindow(l_hDlg);
            l_hDlg = NULL;
            loaded = 0;
        }
        break;
    }
    return TRUE;
}
```

The following sample code creates a gear using the specified parameters. You can copy this code and adapt it for use with your own layout palette. Save it as **gear.c**.

```
void GearMacro ( void )
{
 LPoint    Polygon [ 100 ];
 float    Angle2, Angle3, R2, R3, Tooth_Angle;
 LCoord    R_Inner, R_Outer, Teeth_Count, Teeth_Width,
   Tooth;
 LCell    Cell_Draw  = LCell_GetVisible ( );
 LFile    File_Draw  = LCell_GetFile ( Cell_Draw );
 LPoint    Translation = LCursor_GetPosition ( );
 LDialogItem Dialog_Items [ 3 ] = { { "Inner Radius", "175"
   },
                   { "Outer Radius", "200" },
                   { "Teeth Count ", "15 " } };
do {
 if ( !LDialog_MultiLineInputBox ( “Gear Properties”,
  Dialog_Items, 3 ))
       return;
 R_Inner   = atol ( Dialog_Items [ 0 ].value );
 R_Outer   = atol ( Dialog_Items [ 1 ].value );
 Teeth_Count = atol ( Dialog_Items [ 2 ].value );
 Teeth_Width = 6.283185307 * R_Inner / ( 2 * Teeth_Count );
} while ( ( Teeth_Count < 3 ) || ( Teeth_Count > 25   )
    || ( Teeth_Width < 2 ) || ( R_Inner   >= R_Outer )
    || ( R_Inner   < 2 ) );
for ( Tooth = 0; Tooth < Teeth_Count; Tooth++ ) {
 Tooth_Angle = 6.283185307 * Tooth / Teeth_Count;
 R2 = sqrt ( R_Outer * R_Outer + Teeth_Width * Teeth_Width
  / 4.0 );
 R3 = sqrt ( R_Inner * R_Inner + Teeth_Width * Teeth_Width
  / 4.0 );
 Angle2 = atan2 ( Teeth_Width / 2.0, R_Outer );
 Angle3 = atan2 ( Teeth_Width / 2.0, R_Inner );
 Polygon [ 4 * Tooth + 0 ] = LPoint_Set (
  R3 * sin ( Tooth_Angle - Angle3 ) + Translation.x,
  R3 * cos ( Tooth_Angle - Angle3 ) + Translation.y );
 Polygon [ 4 * Tooth + 1 ] = LPoint_Set (
  R2 * sin ( Tooth_Angle - Angle2 ) + Translation.x,
  R2 * cos ( Tooth_Angle - Angle2 ) + Translation.y );
 Polygon [ 4 * Tooth + 2 ] = LPoint_Set (
  R2 * sin ( Tooth_Angle + Angle2 ) + Translation.x,
  R2 * cos ( Tooth_Angle + Angle2 ) + Translation.y );
 Polygon [ 4 * Tooth + 3 ] = LPoint_Set (
  R3 * sin ( Tooth_Angle + Angle3 ) + Translation.x,
   R3 * cos ( Tooth_Angle + Angle3 ) + Translation.y );
}
LPolygon_New ( Cell_Draw, LLayer_Find ( File_Draw, “Metal1”
    ), Polygon, 4 * Teeth_Count);
LCell_MakeVisible ( Cell_Draw );
LCell_HomeView(Cell_Draw);
}
```

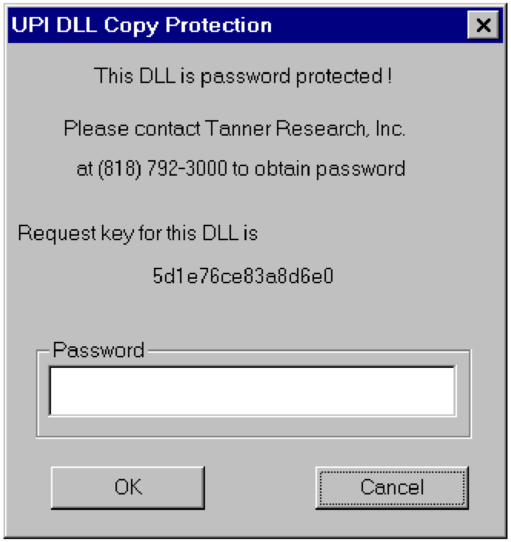
L-Edit comes with source code that you can adapt for use with your own layout palette. The directory **L-Edit Pro\Samples\samples\UPI\DLL\palette** contains the complete source code required to create the layout palette DLL.

Copy-Protecting Macro DLLs

In this lesson, you will learn to use copy-protected UPI macro DLLs and to copy-protect your own macro DLLs. The example used is the UPI macro created in Creating a Compiled Macro (DLL) on page 4-27 which draws an Active-to-Metal contact at the current mouse location. This DLL is copy-protected, and users will have to call the vendor to obtain the proper password.

The following procedure explains how to use a copy-protected DLL:

☑ Using the **Macro** dialog, load the macro **Contact_Active_Metal1**. L-Edit will display the following dialog:


UPI DLL Copy Protection

This DLL is password protected !

Please contact Tanner Research, Inc.

at (818) 792-3000 to obtain password

Request key for this DLL is

5d1e76ce83a8d6e0

☑ Call the DLL vendor and ask for a password.

☑ Provide the vendor with your L-Edit serial number and request a key for the DLL. The vendor will compute the password and authorize the user by providing a function:

```
password = f(L-Edit Serial number, Request key for DLL)
```

where **f()** is a vendor-defined function that returns the password.

☑ Enter the password in the **Password** field. The DLL will compare the password with the internally generated password. If they match, the DLL will be loaded.

If the password succeeds, it will be stored in the system registry. That way, you will not have to retype the password every time you try to load the DLL.

The following procedure explains how to create a copy-protected DLL.

☑ Write code for creating a regular DLL.

☑ Create the outline for DLL copy protection.

☑ Create a password verification dialog.

☑ Write the dialog handling routine for the password verification dialog.

☑ Compile the DLL.

In outline form, the code for DLL copy protection looks like this:

```
#include “ldata.h”
void Contact_Active_Metal1 ( )
{
    <write code for creating contact>
}

int UPI_Entry_Point( void )
{
    <Check for the password. Return zero if unsuccessful.>
    LMacro_BindToHotKey ( KEY_F1, “My Contact,
        Active-Metal1”,     “Contact_Active_Metal1” );
    return 1;
}
```

VerifyDLLPassword() is a user-supplied function that will display a dialog for password entry. If the password is entered correctly, **DllMain()** returns TRUE and the DLL is loaded.

In outline form, the code for the **VerifyDLLPassword** function looks like this:

```
int VerifyDLLPassword( char *dll_file_name )
/* DESCRIPTION: calculate a challenge based on the dll file
   and request the password from the user. If password
   matches with one produced internally, then return 1 else
   return 0*/
{
    <based on dll file create a key (challenge)>

    /* Check if the user has previously entered the password
   */
    password_found_in_registry =
   get_password_from_registry(challenge, pass1);
```

```
    if ( password_found_in_registry != SUCCESS ){
        /* request password from the user */
        if (!GetDLLPassword(challenge, password1))
            return FAILURE;
    }

    <get the L-Edit serial number>

    <generate password internally, and check user's password
   against it
            actual_challenge = f(challenge,L-Edit serial
   number >

<Use actual_challenge to internally calculate the password2>

    if (strcmp(password1, password2) == 0){
        if ( password_found_in_registry == FAILURE )
            store_password_in_registry( challenge, password1
   );
        return SUCCESS;
    }
    else {
         LDialog_AlertBox("Incorrect password!");
        return FAILURE;
    }
}
```