IDRISI

Applications Programming Interface

User's Guide

Version 2.0 © Clark Labs 2003

IDRISI has been designed as an OLE Automation Server using COM Object technology (i.e., a COM Server)[1]. As a consequence, it is possible to use high-level development languages, such as Delphi, Visual C++, Visual Basic, or Visual Basic for Applications (VBA) as macro languages for controlling the operation of IDRISI. In addition, you can create sophisticated OLE Automation Controller applications that have complete control over the operation of IDRISI.

The OLE Automation Server feature of IDRISI is automatically registered with Windows when IDRISI is installed onto your system. Thus, if you have installed IDRISI, you automatically have access to the full API. The IDRISI OLE Automation Server provides a wide range of functions for controlling IDRISI, including running modules, displaying files, and manipulating IDRISI environment variables.

This guide is broken down into several sections:

The first four sections are relevant to all users. Section 5 is of concern only to those who wish to extend the functionality of IDRISI by creating their own modules. This includes information on how to add new items to the menu system, how to have IDRISI monitor and display the progress of the module, and so on. Sections 6,

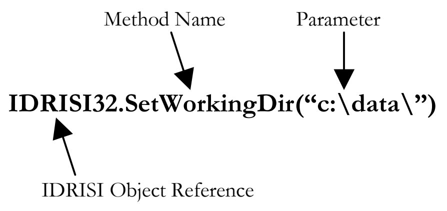
[1] The API works with all 32-bit versions of IDRISI, including IDRISI Release One and Two, and IDRISI Kilimanjaro.

7 and 8 provide examples of how the API can be used. Note that these last three sections are instructive even for those who are not familiar with these programming products.

1. Quick Start

The first step in using the IDRISI API is to register it with your programming environment. This typically takes only a few moments, and is described in Section 2 below. After you have registered the API you then have the ability to control most aspects of the operation of the IDRISI System. Here are some quick tips on getting started:

- Once the IDRISI Library has been installed, any of the included IDRISI methods or properties can be accessed by simply typing the name of the IDRISI *object reference*, followed by a period, followed by the *method* or *property* name and its *parameters*. e.g.:



IDRISI32.SetWorkingDir("c:\data\")

  The IDRISI *object reference* is the name your programming environment uses to refer to the IDRISI Automation Controller. It is established when you install the IDRISI Library (see below). *Methods* are functions or procedures included in the automation controller library (i.e., operations that will cause some action to take place), while *properties* are user-accessible settings (such as the path to the working folder).
- In most programming environments, you will notice that as soon as you finish typing the period after the IDRISI32 object reference, you will be provided with a list of library functions from which you can choose. Likewise, as soon as you type the open parenthesis after the function name, it will list all the parameters required, as well as their data types.
- If you create a program that accesses the API and IDRISI is not currently running, Windows will automatically launch it when needed.
- Section 3 contains a very long list of methods (functions and procedures) and properties (attributes). Most users only need three or four! The most important method is **RunModule**. This is used to launch an IDRISI module, and uses the same list of parameters as a standard IDRISI macro. After this, most will find the **DisplayFile** method to be useful. As the name suggests, it is used to display a raster or vector layer. Note that to add a layer to an existing composition, you should use **AddLayerToDisplay**. After this, most users will only need one or all of the **GetWorkingDir**, **GetResourceDirCount**, and **GetResourceDir** methods. These are used to determine the paths of the current working and resource directories.

- The **RunModule** method has a wait option (that should normally be used) that causes your program to wait until the IDRISI operation has finished before going on to the next line of your code. However, note that the **RunMacro** method, as originally designed, does not have this. As a consequence, it will start the macro, but immediately (i.e., without waiting for the macro to finish) return control to your program. A method named **RunMacroAndWait** has been added that will wait until the macro has finished. Alternatively, you can recode the macro into calls to **RunModule** within your program.
- The ModParams parameter of RunModule is identical to the macro command for the module being run, with two exceptions. First, the "x" parameter of the macro command is not needed. Second, all file references used with **RunModule** should use the full path and extension. For layers, the extension of the main data file should be used (".rst", ".vct", ".avl") rather than that of its corresponding documentation file. For example (using Visual Basic):

```
Dim CmdStr as String,Title as String,Units as String[2]
Dim Success as Integer
CmdStr="5*c:\data\band4.rst*c:\data\band3.rst*c:\data\ndvi.rst"
Title="Normalized Difference Vegetation Index"
Units="NDVI"
Success = IDRISI32.RunModule("Overlay",CmdStr,True,Title,Units,"","",True)
```

As an alternative to hard coding the path, consider using the API methods that access the working and resource folders. For example:

```
Dim CmdStr as String,WrkPath as String,Title as String,Units as String
Dim Success as Integer
WrkPath = IDRISI32.GetWorkingDir
CmdStr = "3*"+WrkPath+"band4.rst*"+WrkPath+"band3.rst*"+WrkPath+"ndvi.rst"
Title="Normalized Difference Vegetation Index"
Units="NDVI"
Success = IDRISI32.RunModule("Overlay",CmdStr,True,Title,Units,"","",True)
```

- You may notice a few methods in the IDRISI Library that are not documented in Section 3. These are for internal use by IDRISI only and should not be used in your programs.
- Also note that you can modify the IDRISI menu. The procedures for doing so are detailed in Sections 5 and 7. This allows you to incorporate menu entries for new modules you create.

[2] Be careful declaring multiple variables with the same Dim statement in Visual Basic. If the data type is not specifically indicated (as in this example) for any variable, it is implicitly declared as a variant. Thus, for example, if this statement had read "Dim CmdStr,Title,Units as String", only Units would be String – the other two variables would be variants!

2. Accessing the IDRISI API

Any development language that supports OLE Automation (such as Delphi, C++, Visual Basic and VBA) usually provides simple facilities to allow you to manipulate IDRISI. Included below are specific instructions for using the API with Visual Basic, Visual Basic for Applications (VBA – found in all Microsoft Office products such as Excel, Access and Word), and Delphi. In other cases, you should look in the help system for the programming environment you are using under the name *Automation Controller*. IDRISI is an Automation Server. Thus any program that accesses these methods and properties is called an Automation Controller.

Accessing IDRISI from VBA (in Word, Access, Excel)

From a Microsoft Office product, such as Word, Access or Excel, start the Visual Basic editor (usually found under the Tools menu). Once you are in the VBA editor, go to the *Tools* menu and click on the *References* menu item. This will launch a dialog in which you will see a list of all OLE Automation Servers registered on your computer. If you have installed IDRISI, you will see a listing for *IDRISI Library*. Click the checkbox beside it to make this server available. Then click on the OK button. You are now ready to begin programming IDRISI.

To access any of the library functions, simply type "IDRISI32" (without the quotes) followed by a period ("."), followed by the function name and its parameters. You will notice that as soon as you finish typing the period after the IDRISI32 key word, VBA will provide a list of library functions from which you can choose. Likewise, as soon as you type the open parenthesis after the function name, VBA will list all the parameters required, as well as their data types. Similarly, VBA lists all IDRISI functions in its *Object Browser*.

Note that some of the variable types used in VBA are slightly different that those listed in the method and property reference below. In VBA, the WideString type is simply String. Similarly the WordBool type is simply Boolean. Use the data types as suggested by VBA in its *Object Browser* or code completion facility – these will always be the correct types.

Accessing IDRISI from Microsoft Visual Basic

Accessing IDRISI from Microsoft Visual Basic is as simple as it is from VBA. For each project that needs to access IDRISI, go to the *Project* menu and then the *References* item. This will launch a dialog in which you will see a list of all OLE Automation Servers registered on your computer. If you have installed IDRISI, you will see a listing for *IDRISI Library*. Click the checkbox beside it to make this server available. Then click on the OK button. You are now ready to access IDRISI in your Visual Basic application.

To access any of the library functions, simply type "IDRISI32" (without the quotes) followed by a period ("."), followed by the function name and its parameters. You will notice that as soon as you finish typing the period after the IDRISI32 key word, Visual Basic will provide a list of library functions from which you can choose.

Likewise, as soon as you type the open parenthesis after the function name, Visual Basic will list all the parameters required, as well as their data types. Similarly, Visual Basic lists all IDRISI functions in its *Object Browser*.

Note that some of the variable types used in Visual Basic are slightly different from those listed in the method and property reference below. In Visual Basic, the WideString type is simply String. Similarly the WoodBool type is simply Boolean. Use the data types as suggested by Visual Basic in its *Object Browser* or code completion facility – these will always be the correct types.

Accessing IDRISI from Delphi

The following information is specific to Delphi Version 5.0. For earlier versions, similar procedures exist. However, you should refer to the help system of your specific version for details.

Step 1 : Installing the IDRISI Type Library (Do this once)

From the PROJECT menu in Delphi, select IMPORT TYPE LIBRARY. The find "IDRISI Library" in the list of installed servers and select it. Then set the palette page to "Servers" and be sure the "Generate Component Wrapper" checkbox is checked "on". Then click on "Install". You will then be asked which package to install it into, with the default being "dclusr50.dpk". This is the main Delphi component package. Click on OK to signify that you do wish to install it to that package. You will then be presented with a dialog about the contents of this library. Now click on "Compile". You can now close this dialog – you will now have access to the IDRISI Type Library in any Delphi project you create. *NOTE : You will need to repeat this process if you install an update to IDRISI32.EXE. This will replace the old type library with the new one.*

Step 2 : Using the IDRISI Type Library (Do this for every new project)

Within any project that needs to access IDRISI, include the following "Uses" statement:

```
Uses IDRISI32_tlb;
```

Then declare an object of the IIdrisiApiServer class as follows:

```
Var IDRISI32 : IIdrisiApiServer;
```

Then in your implementation section, declare the IDRISI API object as follows:

```
IDRISI32:=CoIdrisiApiServer.Create;
```

You can then access any property or method provided by IDRISI by referencing the IDRISI32 object, followed by a period, followed by the method or property name and any required parameters. For example, given a string variable named "Working_Folder":

```
Working_Folder:=IDRISI32.GetWorkingDir;
```

Finally, before allowing your application to terminate, signify to the API that you no longer need services as follows:

```
IDRISI32:=nil;
```

With this last operation you remove your process as a client of IDRISI32. This allows IDRISI32 to later be terminated without concern that an application is present that still needs services.

Accessing IDRISI32 from Microsoft Visual C++

The following instructions apply to the use of the IDRISI32 API with Microsoft Visual C++ Version 6.

1. Go to the FILE | NEW menu option and click onto the PROJECT tab.
2. From the list of "new" options, select the MFC AppWizard (exe) option.
3. Specify the project location and name and then click OK.
4. In Steps 1 - 2 of the AppWizard dialog sequence, accept the defaults (or choose whichever options you need).
5. In Step 3 of the AppWizard sequence there is a question about "What other support would you like to include?" Make sure that the "Automation" option is selected to be on (i.e., it has a check mark).
6. In Steps 4 – 6 of the AppWizard sequence, accept the defaults (or choose whichever options you need). Then click on the Finish and OK options to complete the process of initiating the application.
7. Now choose the VIEW | ClassWizard menu option and choose the Automation tab. Select the "Add Class" button and then the "From a Type Library" option. You will then be presented with a file selection dialog that is designed to look for a ".tlb" file (a type library). However, the Type Library for IDRISI32 is contained in the IDRISI32.EXE executable. Therefore, change the file mask to indicate "Any file (*.*)" and then locate the IDRISI32.EXE main program module in the IDRISI folder. Visual C++ will then extract the header and implementation details of the IDRISI automation server and structure them within standard C++ files: IDRISI32.h and IDRISI32.cpp. Click on OK to accept these details about the specification of the IIdrisiAPIServer class. Then click on OK to exit the AppWizard dialog.
8. To access the IDRISI API, you will then need to register your application as a client of the IDRISI32 COM Server. Then after you have finished, you will need to unregister your application as a client. This is done as follows:

- Declare an object of type IIdrisiAPIServer.
- Call the CreateDispatch method of that object, passing the argument "IDRISI32.IdrisiAPIServer".
- After all calls to the API, call the ReleaseDispatch method . e.g.,

```
IIdrisiAPIServer  IDRISI;                          //create the API object
IDRISI.CreateDispatch("IDRISI32.IdrisiAPIServer");  //call its interface constructor
CString wrkfolder=IDRISI32.GetWorkingDir( );        //an IDRISI API call
IDRISI32.ReleaseDispath( );                         //release the interface reference
```

With this last operation you signify to the server that you no longer need services. This allows IDRISI to be terminated without concern that an application is present that still needs services.

Accessing IDRISI from Python

Python is a portable, interpreted, object-oriented programming language meant to interface with IDRISI through Windows COM Server technology. Programming knowledge is required to use the Python language and syntax with its powerful built in high-level data types. Programs written in Python can be extended in a systematic fashion by adding new modules implemented in a compiled language such as C or C++. Such extension modules can define new functions and variables as well as new object types. Like any other COM compatible programming language, Python is capable of accessing IDRISI's COM Server and to control IDRISI programming facilities.

The advantage of Python is that it is freely available software and can be downloaded from the Python website. Complete details of Python extensive programming examples and help can be found at: www.python.org. As a third-party software, the Clark Labs does not support this product nor is able to provide technical support for any programs developed with Python.

Setting up Python

After installing Python and the Windows Extensions, it is possible to access Python's command interpreter directly from IDRISI or from Windows Start/Programs menu.

The following are some options for selecting Python's command interpreter:

"IDLE (Python GUI)"	(file: idle.pyw*)
"Python win"	(file: Pythonwin.exe)

* used in the examples in section 8.

Python can run in either interactive or programming modes. In interactive mode, Python offers a command line prompt that can be used to call the IDRISI API. In programming mode, Python can run the same commands but they are saved in a file (extension .py) to be run as a Python program. It is also possible to pass parameters to access a graphical user interface for Python programs. More is available on this topic on the Python website.

3. API Methods and Properties

The following is a list of the functions that are available to users. Note that the parameters are listed with the data types as specified in Delphi. For Visual Basic and C++ there are slight differences (which will probably be obvious) – Delphi's WideString type is interpreted by Visual Basic as simply String; similarly the Delphi's WordBool type is interpreted as Boolean in Visual Basic. Likewise, in C++ Delphi's Single is interpreted as Float and Widestring is CString. When in doubt, use the syntax suggested by your language compiler.

For Directory/Project Info:

function GetWorkingDir: WideString;

This function returns the user's current working folder. The user's current working folder name is stored in that user's current project file (.env file). You may need to append a backslash to the result. See also GetResourceDir, Set_WorkingDir, SaveProjectToFile, and Get_CurrentProjectFileName.

function SetWorkingDir(const WorkingDir: WideString): WordBool;

This function sets the default working data path for the current user. Make sure to append a backslash to the end of the WorkingDir! (e.g. SetWorkingDir('c:\newworkingdir\');) . This function currently only changes the working folder in memory! It does not save it after the user exits IDRISI. To change and save a working directory (recommended), call SetWorkingDir and then SaveProjectToFile (using the result of Get_CurrentProjectfilename as input for SaveProjectToFile).

function GetResourceDirCount: Integer; safecall;

In addition to a working directory, a user's project file may also contain any number of resource directories. This function returns the number of resource directories in the current project file. See also GetResourceDir.

function GetResourceDir(ResIndex: Integer): WideString;

This function returns the path of the resource folder (in the current project file) specified by ResIndex. Resource directories are stored by IDRISI in a list. You may obtain the name of any resource directory in the list by using GetResourceDir and specifying the index of that directory in the list. To enumerate all of the Resource directories, call GetReourceDirCount to determine the number of Resource directories, and use GetResourceDir for each directory. Note that the resource dir list is not a zero-based list - it starts at 1.

function AddResourceDirToProject(const ResourceDir: WideString): WordBool;

This function allows you to append a resource folder to the existing list of resource folders. The ResourceDir parameter should include the entire path, including an ending backslash. This function returns true or false depending on whether the operation was successful.
You may also want to look at SetWorkingFolder, SaveProjectToFile, and OpenProjectFromFile.

function RemoveResourceDirFromProject(const ResourceDir: WideString): WordBool;

This function removes the specified resource folder (ResourceDir) from the project. See also AddResourceDirToProject and GetResourceDir.

function RemoveAllResourceDirs: WordBool;

This function removes all resource directories from memory. See also AddResourceDirToProject and RemoveResourceDir.

function Get_CurrentProjectFileName: WideString; safecall;

This function returns the name of the project file (.env file) that is currently active. It returns the full path and filename with extension. If you need to change the working directory, use “setworkingdir”. If you need to switch the entire project file and open another project file, see “Set_DefaultProjectFileName”. If you want to save the current settings first before switching to another project file, use “SaveProjectToFile”.

procedure Set_CurrentProjectFileName(const Value: WideString); safecall;

This procedure sets the name of the current default project. After you set this parameter, you **must** call “OpenFromFile”. If you first want to save the current project file (.env) settings, see “SaveProjectToFile”. This function is the same as Set_DefaultProjectFileName.

function OpenProjectFromFile(const FileName: WideString): WordBool;

This function opens a new project. After calling this function, you need to call SetDefaultProjectFile. For more information about projects, see Get_CurrentProjectFileName.

function SaveProjectToFile(const FileName: WideString): WordBool;

This function saves the current configuration (working folder / resource folders) to the current project file. SetWorkingDir, AddResourceDirToProject, and RemoveResourceDir do not save to file. It is recommended that you call SaveProjectToFile after using any of those functions. For more information about projects, see Get_CurrentProjectFileName.

function Get_ProgramDirectory: WideString;

This function returns the name of the directory in which IDRISI is located. Note that IDRISI must be running while this call is made.

function Get_GeorefDir: WideString;

This function allows you to determine the folder where a user's georeferencing files are stored. Full path is returned (including the last backslash). Note that although the result is the typical repository for georef files, these files can be also stored anywhere, including in the working path. (This situation is more likely when the user is running a networked version of IDRISI.)

function Get_SymbolDir: WideString; safecall;

This function returns the name of the symbol file folder (e.g., “c:\IDRISI Kilimanjaro\symbols\”). Note that although most users will store their symbol files in this folder, symbol files from any location (including the working path) can be used by IDRISI (this is particularly likely on a networked version of IDRISI.).

function Get_WavesDir: WideString; safecall;

This function returns the name of the Waves folder (e.g., “c:\IDRISI Kilimanjaro\waves\”).

function Get_CartaLinxPath: WideString;

This function retrieves the CartaLinx path, which is stored in the idrusers.ini file. Note that some users may not CartaLinx, or may not have edited this setting.

procedure Set_CartaLinxPath(const Value: WideString);

This procedure sets the CartaLinx path in the idrusers.ini file. Full path and filename are required.

function Get_BackgroundColor: WideString;

This function retrieves the background color, which is stored in the idrusers.ini file. Note that the background color can be set in user preferences in IDRISI.

procedure Set_ BackgroundColor (const Value: WideString);

This procedure sets the background color path in the idrusers.ini file. Note that the background color can be set in user preferences in IDRISI.

To Run Modules or Processes:

function Run_Macro(const MacroFile, Parameters: WideString): Integer;

Run macro, like the Run Macro dialog box, allows you to call a macro file (with optional parameters). MacroFile is the name of the macrofile (including full path and extension). Parameters are only necessary if variables are used in the macro file. Variables are identified by the percent sign and a number (such as %2) Each parameter is separated by a space. Look in the help system under 'Macro' for more information.

function RunModule(const ModName, ModParams: WideString;
WaitFlag: WordBool; const Title, Units, Extra1, Extra2: WideString;
ProcAutoRemove: WordBool): Integer;

This function calls an IDRISI module using command-line syntax. (Modules which can by run by command line include modules in the IDRISI menu that are in all caps. Also, IDRISI File Explorer functions such as copy, rename, and delete can be run using RunModule. Look at the command-line information under IDRISI File Explorer's Help).

Parameters:

1. Modname: Module names and parameters for each module are provided in the help system.
2. Modparams: must contain the macro parameters specific to that module. These parameters are specified in the help system. A quick way to get the module name and parameters is to call a module from the IDRISI interface, fill out the desired settings, and then copy this information from the IDRISI32.log file (located in the working directory).
3. Waitflag: determines whether the function will wait until the module is finished before your program is to proceed.
4. Title: title for output file (where applicable)
5. Units: value units for the output file (where applicable).
6. Extra1: not used. Send an empty string: ''.
7. Extra2: not used. Send an empty string: ''.
8. ProcAutoRemove should be set to true.

RunModule will automatically report the progress of IDRISI modules in the progress status bar. You do not need to call FreeProcess or TerminateProcess when RunModule is finished.

RunModule returns a Process ID. You can check on the state of the module by calling GetProgress using the process ID.

function RunProcess(const modname, modparams: WideString;
waitflag: WordBool; const title, units, extra1, extra2: WideString;
ProcAutoRemove: WordBool; const DisplayParams: WideString): Integer;

This function is similar to RunModule, but will automatically bring up the results when finished (IF the user's UseAutoDisplay setting is set to true). See RunModule for information about the first 8 parameters.

DisplayParams is a widestring containing display parameters separated by the asterisk character ('*'). The parameters contained within DisplayParams are as follows:

1. Autodisplay the result? Set this to either "true" or "false".
2. Filename (path and extension not necessary) of the file to be displayed.

3. Symbol filename (path and extension not necessary)
4. Layer type: “0”=image, “1”=vector.
5. Autoscale? “0”= don’t autoscale,”1”= autoscale. (If the image is real, it will ignore this parameter and autoscale the image.)
6. Show the title? “1”=show title, “2”=don’t show title, “0”=use current user’s default settings for title.
7. Show the legend? “1”=show legend, “2”=don’t show legend, “0”=use current user’s default settings for legend.
8. Reserved. This parameter must be “0”
9. The final parameter must be “true”.

Example of displayparams string: “true*h87tm1*idris256*0*0*0*0*0*true”

function DisplayFile(const FileName, symbolname: WideString; LayerType,
ScaleType, Title, Legend, Reserved: Integer; SilentLaunch: WordBool;
const mapwindowname: WideString): Integer;

This function displays FileName using the IDRISI display system. Filename should be the name of a raster, vector, or map composition file. This filename can either be a short name (no path, no extension) if the file is located in the current working or resource folder - or a fully - qualified filename. (Use the .rst, .vct, or .map extension). The symbolname, likewise, can be a short name if it is located in the IDRISI Kilimanjaro\symbols folder in a working or resource folder. Symbol files must match the type of layer that will be displayed. Raster files require a .smp file; vector files require a symbol file based on the type of layer (.sm0 = point, .sm1=line, .sm2=polygon). For vector layers, if you specify “default” with no extension, IDRISI will choose the appropriate default symbol file based on that layer. Map compositions do not require a symbol file, so you may leave the symbolfile variable empty in this case.

LayerType is an integer variable which specifies the type of layer you are displaying. The values are as follows.

0=image file

1= vector file

2=map composition file (not implemented yet)

ScaleType determines whether the file will be autoscaled. 0=no autoscaling, 1=autoscaling.

The Title and Legend parameters instruct IDRISI whether or not to show the title and legend. In both cases, 0=use current user’s default settings, 1=true, and 2=false.

The Reserved parameter is not currently used; this value should be set to zero.

SilentLaunch should always be set to true.

The MapWindowname should be the same as FileName.

function AddLayerToDisplay(const filename, symbolname: WideString;
layertype, scaletype, nclasses: Integer): Integer;

This function adds a layer to a currently displayed window. Filename should be the name of a raster, vector, or map composition file. This filename can either be a short name (no path, no extension) if the file is located in the current working or resource folder - or a fully - qualified filename. (Use the .rst, .vct, or .map extension). The symbolname, likewise, can be a short name if it is located in the IDRISI Kilimanjaro\symbols folder in a working or resource folder. Symbol files must match the type of layer that will be displayed. Raster files require a .smp file; vector files require a symbol file based on the type of layer (.sm0 = point, .sm1=line, .sm2=polygon). For vector layers, if you specify "default" with no extension, IDRISI will choose the appropriate default symbol file based on that layer. Map compositions do not require a symbol file, so you may leave the symbolfile variable empty in this case.
LayerType is an integer variable that specifies the type of layer you are displaying. The values are as follows.
0=image file
1= vector file
2=map composition file (not implemented yet)
ScaleType determines whether the file will be autoscaled. 0=no autoscaling, 1=autoscaling.
NClasses determines the number of classes to be displayed. Values range from 0 to 255

function AllocateProcess: Integer;

If you intend to use IDRISI's progress reporting, you must call AllocateProcess. This function informs IDRISI that a process is starting. It then returns a process ID for that process. The process ID can be subsequently used for progress reporting. (such as NotifyWorking, ProportionDone, SimplePass, PassInProgress, and PassProportion) If you use AllocateProcess, you must call ProcessFinished, then FreeProcess when your task is finished. These functions will ensure that your process cleans up after itself. If you want to abort the process, call ProcessTerminated (instead of ProcessFinished) and then FreeProcess. AllocateProcess is usually followed by a call to Set_Process_ModuleName.

function Set_Process_ModuleName(ProcID: Integer; const modulename: WideString): Integer;

When you do AllocateProcess, you get a ProcId, and IDRISI begins to report your module's progress, but it doesn't know the name of your module. On the status bar, or in the progress box, your module will not have a name unless you use this function to set it.

function FreeProcess(ProcId: Integer): Integer;

If you have created your own process (using AllocateProcess), you must call this function when your process has finished running. This lets IDRISI know to free the process ID and stop showing progress reports for that process. Call ProcessFinished or ProcessTerminated before calling FreeProcess. See AllocateProcess for more information.

function RaiseError(ProcId, ErrorCode: Integer; const Extra1,
Extra2 : WideString): Integer;

If you are reporting the progress status of your process (see AllocateProcess), and encounter an error during your process, you can call RaiseError to report this error through IDRISI. The ProcID is the process ID derived from your AllocateProcess call. The error code corresponds to an IDRISI error code (see english.err). Extra1 and Extra2 might be required for the error code message. External applications might find it more useful to create their own error messages (be sure to use ProcessTerminate and FreeProcess, however).

function RaiseWarning(ProcId, WarningCode: Integer; const Extra1, Extra2: WideString): Integer;

If you are reporting the progress status of your process (see AllocateProcess), and want to raise a warning message (without ending your process), you can call RaiseWarning to bring up the warning message. The ProcID is the process ID derived from your AllocateProcess call. The error code corresponds to an IDRISI error code (see english.err). Extra1 and Extra2 might be required for the error code message. External applications might find it more useful to raise their own warnings.

function ProportionDone(ProcId: Integer; Proportion: Single): Integer;

If you would like to report the progress status of your process (see AllocateProcess), this function changes the IDRISI progress status bar. Progress reporting can exist in several formats: ProportionDone, SimplePass, PassInProgress, PassProportion, and NotifyWorking.

ProportionDone will show a percentage on the status bar. (for example, “84%”). You must provide the ProcessID (from AllocateProcess) and the proportion (in decimal format; for example “.84”).

Make a call to this function each time you want to update the status bar with the progress of your module.

function SimplePass(ProcId, CurrentPass: Integer): Integer;

If you would like to report the progress status of your process (see AllocateProcess), this function changes the IDRISI progress status bar. Progress reporting can exist in several formats: ProportionDone, SimplePass, PassInProgress, PassProportion, and NotifyWorking.

SimplePass will show a pass number on the status bar (for example, “Pass 3”). You must provide the ProcessID (from AllocateProcess) and the current pass number (An integer; in this instance “3”).

Make a call to this function each time you want to update the status bar with the progress of your module.

function PassInProgress(ProcId, CurrentPass, MaxPasses: Integer): Integer;

If you would like to report the progress status of your process (see AllocateProcess), this function changes the IDRISI progress status bar. Progress reporting can exist in several formats: ProportionDone, SimplePass, PassInProgress, PassProportion, and NotifyWorking.

PassInProgress will show a pass number (out of a given number of maximum passes) on the status bar (for example, “Pass 3 of 4”). You must provide the ProcessID (from AllocateProcess) the current pass number (An integer; in this instance “3”), and the maximum number of passes (an integer; in this instance, “4”).

Make a call to this function each time you want to update the status bar with the progress of your module.

function PassProportion(ProcId, CurrentPass, MaxPasses: Integer;
Proportion: Single): Integer;

If you would like to report the progress status of your process (see AllocateProcess), this function changes the IDRISI progress status bar. Progress reporting can exist in several formats: ProportionDone, SimplePass, PassInProgress, PassProportion, and NotifyWorking.

PassInProgress will show a pass number (out of a given number of maximum passes) on the status bar, as well as the percentage finished for that pass (for example, “Pass 3 of 4, 20%”). You must provide the ProcessID (from AllocateProcess) the current pass number (An integer; in this instance “3”), the maximum number of passes (an integer; in this instance, “4”), and the proportion done (in decimal format; in this instance, “.20”).

Make a call to this function each time you want to update the status bar with the progress of your process.

function GetProgress(ProcId: Integer; var ProgressType, ErrorCode,
WarningCode: Integer; var Proportion: Single; var CurrentPass,
MaxPasses: Integer): Integer;

This function allows you to query the progress of a specified process. For example, if you have used RunModule (or RunProcess) to run an IDRISI module, you can use that processID to query the progress of the module.

process status codes
IDR_STATUS_FINISHED = 30;
IDR_STATUS_ERROR = 31;
IDR_STATUS_PROPORTIONDONE = 32;
IDR_STATUS_SIMPLEPASS = 33;
IDR_STATUS_PASSINPROGRESS = 34;
IDR_STATUS_PASSPROPORTION = 35;
IDR_STATUS_WORKING = 36;

IDR_STATUS_UNINITIALIZED = 37;

IDR_STATUS_WARNING = 38;

IDR_STATUS_TERMINATED = 40;

function NotifyWorking(ProcId: Integer): Integer;

If you would like to report the progress status of your process (see AllocateProcess), this function changes the IDRISI progress status bar. Progress reporting can exist in several formats: ProportionDone, SimplePass, PassInProgress, PassProportion, and NotifyWorking.

Notify working will simply read "Working…" on the status bar. You must provide the ProcessID (from AllocateProcess).

Make a call to this function each time you want to update the status bar with the progress of your module.

function ProcessFinished(ProcId: Integer): Integer;

This function informs the progress reporting that a process (see AllocateProcess) has completed successfully. If your module finishes (and does not RaiseError or ProcessTerminated), call this function, and then call FreeProcess. It informs IDRISI that your process has finished cleanly. This is particularly important if you are auto-displaying your output; the auto-display checks to see if a process has called this ProcessFinished function. Note: If the process has finished unsuccessfully, call ProcessTerminated instead of ProcessFinished.
See AllocateProcess for more information.

procedure ProcessTerminated(ProcID: Integer);

ProcessTerminated informs IDRISI that a process (started by AllocateProcess) has finished, but has been unsuccessful. It is important to call ProcessTerminated rather than ProcessFinished if the operation is unsuccessful - particularly in those cases in which ProcessFinished would have displayed a result.

To get or set SystemInfo (project settings):

function Get_MainWindowState: Integer;

This function retrieves the current state of the main IDRISI window. It should return one of the following values:

Normal=1

Minimized=2

Maximized=3

procedure Set_MainWindowState(Value: Integer);

This procedure will manipulate the state of the main IDRISI window interface. The values are as follows:

1 = Normal

2 = Minimized

3 = Maximized

function GetIniString(const FName, Section, Key: WideString): WideString;

This function calls the Windows API function GetPrivateProfileString. Fname is the name of the file to open. Specify the Section and Keyname of the value that you would like to obtain. This function is not IDRISI-specific.

function SetIniString(const Fname, Section, Key, NewString: WideString): WordBool; safecall;

This function calls the Windows API function WritePrivateProfileString. Fname is the name of the file to edit. Specify the Section and Key of the value (NewString) that you would like to change. This function is not IDRISI-specific.

function Get_INIfilename: WideString;

This function retrieves the path and name of the IDRISI initialization file.

function Get_CurrentProfileName: WideString;

This function returns the name of the current IDRISI user.

function Get_UserINIfilename: WideString;

This function returns the name of the IDRISI initialization file containing the user names.

function Get_HelpFileName: WideString;

Get_HelpFileName returns the name of the IDRISI help file. This file is located in the IDRISI program folder. When IDRISI is run for the first time, the help file is copied into the program folder.

function Get_Language: WideString;

This function allows you to determine the current user's language preference for IDRISI. As of early 2000, English is the only language pack available for IDRISI, so this function will always return "English".

procedure Set_Language(const Value: WideString);

Sets the current user's default language setting for IDRISI. The Value parameter should contain a string with the name of the language. Note: The only language pack currently available is 'English'.

function Get_LangFileName(const section: WideString): WideString; safecall;

This function returns the name of the IDRISI language file.

function Get_ImageDocExtension: WideString;

This function returns the file extension for image documentation files. In IDRISI, this function should always return ".rdc".

function Get_ImageExtension: WideString;

This function returns the file extension for image files. In IDRISI, this function should always return ".rst".

function Get_ValuesDocExtension: WideString;

This function returns the file extension for attribute values documentation files. In IDRISI, this function should always return ".vdc".

function Get_ValuesExtension: WideString;

This function returns the file extension for attribute values files. In IDRISI, this function should always return ".avl".

function Get_VectorDocExtension: WideString;

This function returns the file extension for vector documentation files. In IDRISI, this function should always return ".vdc".

function Get_VectorExtension: WideString;

This function returns the file extension for vector files. In IDRISI, this function should always return ".vct".

function Get_OverwriteProtection: WordBool;

This function determines whether the user has asked to be alerted if a file exists before overwriting it. If Get_OverwriteProtection is true, in most instances IDRISI will bring up a warning message asking whether or not to overwrite the file. You might want to incorporate the same logic into your program.
Note: when running IDRISI from command line parameters, the overwrite protection will be turned off.

function Get_AutoOutputPrefix: WideString;

This function returns the prefix that is used to automatically generate temporary files (in IDRISI the default is "tmp"). See also GetNextAutoFileName.

procedure Set_AutoOutputPrefix(const Value: WideString);

Use this function to set the prefix for the automatically generated files. See also GetNextAutoOutputName. Any changes to this setting will be saved permanently.

function Get_DefaultQualPal: WideString;

This function returns the name of the user's default qualitative palette. Note: this name may or may not have a path and extension attached to it, depending where the user has chosen to store it.

procedure Set_DefaultQualPal(const Value: WideString);

This procedure sets the user's default qualitative palette preference. If the palette is in the IDRISI Kilimanjaro/symbols directory, or in a working or resource directory, you do not need to specify a path or extension. Any changes to this setting will be saved permanently.

function Get_DefaultQuantPal: WideString; safecall;

This function returns the name of the user's default quantitative palette. Note: this name may or may not have a path and extension attached to it, depending where the user has chosen to store it.

procedure Set_DefaultQuantPal(const Value: WideString);

This procedure sets the user's default qualitative palette preference. If the palette is in the IDRISI Kilimanjaro/symbols directory, or in a working or resource directory, you do not need to specify a path or extension. Any changes to this setting will be saved permanently.

function Get_UseAutoDisplay: WordBool;

This function determines whether the user has chosen to automatically display the output of analytical modules. DisplayFile ignores this setting. Note: If you want to automatically display the output of a module, use RunProcess instead of RunModule – but this UseAutoDisplay setting must be set to true!

procedure Set_UseAutoDisplay(Value: WordBool);

This procedure sets a user's preference for automatically displaying the output of analytical modules. (DisplayFile ignores this setting.) Any changes to this setting will be saved permanently.

function Get_AutoShowLegend: WordBool;

This function returns true or false depending on whether the user has chosen to automatically display the legend during autodisplay functions. (DisplayFile ignores this setting.)

procedure Set_AutoShowLegend(Value: WordBool);

This procedure sets the user's AutoShowLegend setting. This setting determines whether or not to display the legend in the autodisplay of analytical modules.

function Get_AutoShowTitle: WordBool;

This function returns true or false depending on whether the user has chosen to automatically display the title during autodisplay functions. (DisplayFile ignores this setting.)

procedure Set_AutoShowTitle(Value: WordBool);

This procedure sets the user's AutoShowTitle setting. This setting determines whether or not to display the title in the autodisplay of analytical modules.

Utilities:

function ShowTextResults(const Description, TextFileName: WideString): Integer; safecall;

This function brings up a 'results box' in IDRISI. If your module has produced a text file which you would like to display, call this module with the name (including full path and extension) of the text file you would like to display. The Description parameter is the string which will come up in the banner of the results box window.

The advantage of the using the results box (as opposed to edit) is that it has buttons which allow you to save to file, save to clipboard, and send results to printer. This function returns the handle of the results box.

function GetFromDocFile(const filename: WideString; ParamType: Integer;
var ErrorCode: Integer): WideString; safecall;

This function allows you to access any element from an image documentation file. Parameters are:

- FileName : the image documentation file name
- ParamType : a numeric code for the parameter desired. Codes listed below:
- ErrorCode : a return error code (also used to specify additional parameters). A return value of 0 signifies success, while negative values indicate that an error has occurred).

Codes for paramType:
1 = Title
2 = File type (0=ASCII / 1 = binary)
3 = Data type
return values for data type:
Integer=0
Real=1
Byte=2
RGB24=3
RGB8=4
String=5

4 = Columns

5 = Rows

6 = Reference system

7 = Reference units

8 = Unit distance

9 = Minimum X

10 = Maximum X

11 = Minimum Y

12 = Maximum Y

13 = Minimum cell value

14 = Maximum cell value

15 = Value units

16 = Number of legend categories

17 = Legend caption (specify the legend category by means of the errorcode field)

18 = Legend code (specify the legend category by means of the errorcode field)

19 = Position error

20 = Reserved

21 = Resolution

22 = Reserved

23 = Value Error

24 = Reserved

25 = Flag value

26 = Flag definition

27 = Reserved

28 = Number of comment lines

29 = Comment line (specify the specific line by means of the errorcode field)

30 = Number of lineage lines

31 = Lineage line (specify the specific line by means of the errorcode field)

32 = Number of consistency lines

33 = Consistency line (specify the specific line by means of the errorcode field)

34 = Number of completeness lines

35 = Completeness line (specify the specific line by means of the errorcode field)

36 = Reserved

37 = Reserved

38 = Metadata version

39 = Display minimum

40 = Display maximum

Note that all results passed to this function are sent as widestrings. It is up to the user to convert the string (such as to an integer) as necessary.

function GetFromVecDocFile(const FileName: WideString; ParamType: Integer;
var ErrorCode: Integer): WideString; safecall;

This function allows you to access any element from a vector documentation file. Parameters are:

- FileName : the vector documentation file name
- ParamType : a numeric code for the parameter desired. Codes are listed below.
- ErrorCode : a return error code (also used to specify additional parameters). A return value of 0 signifies success, while negative values indicate that an error has occurred).

Codes for paramType:
1 = Title
2 = File type (0=ASCII / 1=binary)
3 = Reserved
4 = Reserved
5 = Reserved
6 = Reference system
7 = Reference units
8 = Unit distance
9 = Minimum X
10 = Maximum X
11 = Minimum Y
12 = Maximum Y
13 = Minimum cell value
14 = Maximum cell value
15 = Value units
16 = Number of legend categories
17 = Legend caption (specify the legend category by means of the errorcode field)
18 = Legend code (specify the legend category by means of the errorcode field)
19 = Position error
20 = Reserved
21 = Resolution
22 = Reserved
23 = Value Error
24 = Reserved
25 = Flag value

26 = Flag definition

27 = Reserved

28 = Number of comment lines

29 = Comment line (specify the specific line by means of the errorcode field)

30 = Number of lineage lines

31 = Lineage line (specify the specific line by means of the errorcode field)

32 = Number of consistency lines

33 = Consistency line (specify the specific line by means of the errorcode field)

34 = Number of completeness lines

35 = Completeness line (specify the specific line by means of the errorcode field)

36 = ID type (1=integer / 2=real)

37 =Feature type (geographic object type). 1=point, 2=line, 3=polygon, 4=text

38 = Metadata version

39 = Display minimum

40 = Display maximum

function GetFromValDocFile(const FileName: WideString; ParamType, FieldNum: Integer; var ErrorCode: Integer): WideString;

This function allows you to access any element from a vector documentation file. Parameters are:

- FileName : the attribute values documentation file name
- ParamType : a numeric code for the parameter desired. Codes are listed below.
- FieldNum : the field number from which the metadata element should be retrieved.
- ErrorCode : a return error code (also used to specify additional parameters). A return value of 0 signifies success, while negative values indicate that an error has occurred).

Codes for ParamType:

1 = Title

2 = File type (0=ASCII / 1 = binary / 2= fixed ASCII / 3= ACCESS)

3 = Data type

Return codes:

Integer=0

Real=1

Byte=2

RGB24=3

RGB8=4

String=5

Access Long Integer=6

Access Short Integer=7

Access Single Precision Real=8

Access Double Precision Real=9

Access Text=10

4 = Minimum cell value

5 = Maximum cell value

6 = Value units

7 = Number of legend categories

8 = Legend caption (specify the legend category by means of the errorcode field)

9 = Legend code (specify the legend category by means of the errorcode field)

10 = Value Error

11 = Reserved

12 = Flag value

13 = Flag definition

14 = Reserved

15 = Number of comment lines

16 = Comment line (specify the specific line by means of the errorcode field)

17 = Number of lineage lines

18 = Lineage line (specify the specific line by means of the errorcode field)

19 = Number of consistency lines

20 = Consistency line (specify the specific line by means of the errorcode field)

21 = Number of completeness lines

22 = Completeness line (specify the specific line by means of the errorcode field)

23 = Reserved

24 = Reserved

25 = Metadata version

26 = Display minimum

27 = Display maximum

function AddToLogFile(const logstr: WideString): WordBool;

This function adds logstr to the IDRISI32.log file. The log file is located in the current working directory. This log file is for informational purposes only – there is no required format for logstr.

function ReadVLXFile(const FileName: WideString;
var FileList: WideString): WordBool;

This function reads a VLX (vector link file) and outputs a list of databases [?] associated with that link file. If the function fails, it will return a result of false.

function GetNextAutoFileName: WideString;

This function will generate a default output file name. The automatically generated filenames are a concatenation of the default output prefix (usually tmp) and a sequential three-digit number (between 000 and 999). **function CalculateRasterMinMax(const filename: WideString; fileformat, datatype: Integer; var min, max: Double): Integer;**

This function calculates the minimum and maximum value for a given raster file. The filename must include complete path and extension (.rst).

Fileformat values: (look at .rdc documentation file (or use GetFromDocFile) to determine the format of your file.)
1=binary
2= packed

Datatype values: (look at .rdc documentation file (or use GetFromDocFile) to determine the datatype of your file.)
0=integer
1=real
2=byte

the min and max parameters will return the min and max values of the raster file.

This function will return –1 if unsuccessful, otherwise it will return zero.

For RGB24 data, see CalculateRGB24MinMax.

function CalculateVectorMinMax(const filename: WideString; datatype: Integer; var min, max: Double): Integer;

This function calculates the minimum and maximum value for a given vector file. The filename must include complete path and extension (.vct).

Datatype values: (look at .vdc documentation file (or use GetFromVecDocFile) to determine the datatype of your file.)
0=integer
1=real
2=byte

the min and max parameters will return the min and max values of the vector file.

This function will return –1 if unsuccessful, otherwise it will return zero.

function CalculateAVLMinMaxRecs(const filename: WideString;
datatype: Integer; var min, max: Double;
var val_recs: Integer): Integer;

This function calculates the minimum and maximum value for a given attribute values file. The filename must include complete path and extension (.avl).

Datatype values: (look at .vdc documentation file (or use GetFromValDocFile) to determine the datatype of your file.)

2=string
[need others]
the min and max parameters will return the min and max values of the vector file.

val_recs= the number of records (lines) in the values file.

This function will return –1 if unsuccessful, otherwise it will return zero.

function CalculateResolution(cols, rows: Integer; MinX, MaxX, MinY,
MaxY: Double; var errcode: Integer): Double;

This function calculates the resolution of a pixel, given the number of columns, number of rows, and minimum and maximum X and Y. If errcode = 0, the function returns the result.

function VectorFileToTextFile(const vctfile, txtfile: WideString): Integer;

This function converts a vector file to a vector export format file. The first parameter, vctfile, is the name of the vector file (path and extension are optional). The second parameter, txtfile, is the name of the output text file with extension (path is optional).

function CallVCTtoVXP(const vctfile, txtfile: WideString): WordBool;

This function converts an IDRISI vector file to a text file.

function CallVXPtoVCT(const txtfile, vctfile: WideString): WordBool;

This function converts a text file to an IDRISI vector file.

function CalculateRGB24minmax(const datafilename: WideString; var redmin,

redmax, greenmin, greenmax, bluemin, bluemax: Integer;
var combinedmin, combinedmax: WideString): Integer;

This function calculates the minimum and maximum red green and blue values and the combined minimum and maximum values for RGB24 raster data. Datafilename should be the full name of the raster file, with path and extension (.rst). This function will return a zero value if it is successful.

function CalcVecMinMaxXY(const vecfilename: WideString; var minimumX,
maximumX, minimumY, maximumY: Double): Integer;

This function calculates the minimum and maximum X and Y boundaries for a given vector file. Vecfilename should be the full path and filename (with .vct extension).
A result of zero means that the operation was successful.

4. Using the API for Scripting Macros

This is the simplest possible use of the API. In this case, one is primarily making calls to the RunModule, RunMacro and DisplayFile methods. In addition, it is common to access one or more of the folder properties and methods. Section 6 provides an illustration of this type of application.

Important Notes

- If IDRISI is not running when the script is run, IDRISI will be loaded automatically.
- Even if you are developing what seems to be a very simple macro, there are numerous pitfalls to developing a clean macro script. If your macro uses loops, you should probably develop and test the macro with only very simple loops (e.g., one iteration). Once you are certain that the logic works, then change the number of iterations to reflect the actual number of loops desired.

5. Developing and Integrating New Modules

IDRISI has been especially designed to allow the user to extend its capabilities through the addition of new modules. The menu is fully customizable, allowing you to add in your module as if it were a native element of the IDRISI system. Here follow some brief instructions on how to integrate your module with IDRISI.

Modifying the Menu Structure

The IDRISI menu is not hard coded within IDRISI32.EXE, but rather, is created from a group of menu script files (with an **.Imn** extension) that are located in the **\IDRISI Kilimanjaro\Extensions** folder. As a consequence, the menu is completely extensible and reconfigurable.

Organizing these menu script files is a master file named **Extensions.Lst**. **Extensions.Lst** lists the menu script files, and their order of appearance, in the menu. Each of the script files is an ASCII (text) file with an **.Imn** (*IDRISI Menu*) extension that contains the menu entries and the corresponding DLL or EXE calls[3]. When a user clicks on a menu item, IDRISI will call the DLL or EXE function given in the menu script file.

The structure of the **Extensions.Lst** is a simple list in ASCII (text) format. Each line lists a menu script (**.Imn)** file (without its extension) in the order required to produce the final menu. The default **Extensions.Lst** that comes with IDRISI contains the following:

```
SYSTEM
ANALYSIS
STATS
IMGPROC
DECISION
CHANGE
DATAEDIT
SURFANAL
HELP
```

The first 2 lines of an **.Imn** file give general information about the extension:

```
EXTENSIONNAME   Basic Analysis                    //Must be set
HELPFILE        ANALYSIS.HLP                      //use "" if no help file
```

The next section contains the *Action List*. The name to the left of the equal (=) sign is the *Action Name*. To the right of the equal sign is the name of the DLL or EXE, followed by the function name to call, followed by an optional parameter. In the case of standard IDRISI dialogs, this last parameter is a number which refers to the particular dialog within a DLL. Some of the dialogs are contained in the main executable (IDRISI) and are referenced as such. Otherwise, a specific DLL or EXE is referenced. For example:

```
BEGINACTIONLIST
 ANALYSIS_EDIT=IDRISI32,Bringup_Edit
//Database Query
 ANALYSIS_RECLASS=ANALYSIS.DLL,LaunchExtensionModule,5000
 ANALYSIS_OVERLAY=ANALYSIS.DLL,LaunchExtensionModule,5001
 ANALYSIS_CROSSTAB=STATS.DLL,LaunchExtensionModule,2005
```

[3] For most IDRISI modules, the action invoked by the menu entry calls a dialog contained in a DLL. This dialog subsequently calls an EXE file that executes the analytical operation specified.

```
//Mathematical Operators
 ANALYSIS_SCALAR=ANALYSIS.DLL,LaunchExtensionModule,5015
 ANALYSIS_TRANSFORM=ANALYSIS.DLL,LaunchExtensionModule,5016
 ANALYSIS_IMAGECALCULATOR=IDRISI32,BringUp_ImageCalc
ENDACTIONLIST
```

In this example, the Image Calculator and Edit operations are contained in the main IDRISI executable while the others are contained in extensions DLL's.

When integrating a user-developed module, the action list only needs to contain the name of the executable (which must be found in the IDRISI Kilimanjaro\MODS folder). e.g.:

```
        ANALYSIS_IMAGEOMATIC=IMAGEOMATIC.EXE
```

After the actionlist is the description of the order and location of the menu items themselves. The menu is a hierarchical structure, with submenus identified by the word branch. MenuBreak tells it to insert a separator line. BEGINMENU starts this section; ENDMENU completes it. Any BEGINBRANCH must have an ENDBRANCH. Here is a shortened version of the last section of the ANALYSIS.IMN file:

```
BEGINMENU

  INSERTIONDEPTH 1       //use zero if top level menu item
  IdrMenuAnalysis        //caption of menu item to be inserted under

  ITEMNAME  ANALYSISMENUDatabaseQuery
  ITEMCAPTION Database Query //caption of first item
  ACTION 0          //this one does no action so its action code is zero
  HELPCONTEXT 0     //'0' means not yet linked

  BEGINBRANCH
   ITEMNAME  ANALYSISMENUReclass  //Any unique name that makes sense
   ITEMCAPTION RECLASS       //This is what appears in the menu
   ACTION ANALYSIS_RECLASS   //This refers to the action list above
   HELPCONTEXT 0

   ITEMNAME  ANALYSISMENUOverlay
   ITEMCAPTION OVERLAY
   ACTION ANALYSIS_OVERLAY
   HELPCONTEXT 0
   ITEMNAME  ANALYSISMENUCrosstab
   ITEMCAPTION CROSSTAB
   ACTION ANALYSIS_CROSSTAB
   HELPCONTEXT 0

   MENUBREAK

   ITEMNAME  ANALYSISMENUEdit
   ITEMCAPTION Edit
   ACTION ANALYSIS_EDIT
   HELPCONTEXT 0
  ENDBRANCH


  ITEMNAME  ANALYSISMENUMathematicalOperators
  ITEMCAPTION Mathematical Operators
  ACTION 0
  HELPCONTEXT 0

  BEGINBRANCH
   ITEMNAME  ANALYSISMENUOverlay2
   ITEMCAPTION OVERLAY
   ACTION ANALYSIS_OVERLAY
   HELPCONTEXT 0

   ITEMNAME  ANALYSISMENUScalar
   ITEMCAPTION SCALAR
   ACTION ANALYSIS_SCALAR
   HELPCONTEXT 0
  END BRANCH
ENDMENU
```

Communicating with IDRISI

Through the IDRISI API, you can tell IDRISI that your process is running and have it report the progress of your module as it does for standard IDRISI modules. Below we list a description of the functions that you are most likely to use to make your module communicate with IDRISI.

First you use the **function AllocateProcess** to let IDRISI know that you have a process that will be reporting its progress. AllocateProcess returns an integer value, which is the ProcessID. You will need the ProcessID for the other progress reporting functions.

You then call **Set_Process_ModuleName** to set the module name (for the ProcessID you were just given) to be displayed on the status bar.

As your process runs, you can report its progress with one of the following procedures: **Proportion Done** (percentages), **SimplePass** (e.g. pass 1, pass 2, etc.), **PassInProgress** (e.g. Pass 1 of 3), **PassProportion** (Pass 3 of 4, 20%), or **NotifyWorking** (working…). The progress reports for your process will appear on the status bar and in the Progress Box (if the user has it open).

If your process finishes successfully, call **ProcessFinished** to notify the progress reporting to clear out that module, then **FreeProcess**, to free up the ProcessID.

If your process has an error, you can use our error reporting procedures: **RaiseError** or **RaiseWarning**, or you can show your error messages with your own procedures (e.g. MessageBox). If you use our methods for error handling, the error will be recorded in the IDRISI32.log file. However, you will need to add your error message to english.err or english2.err. We recommend that you just handle the errors yourself, since the language files will be constantly changing and if you try to alter them and distribute an altered file, you may not have the most up-to-date language files.

If your program finishes prematurely because of an error, call **ProcessTerminated** instead of **ProcessFinished,** then call **FreeProcess**. (ProcessFinished would work fine though.)

In addition to running your own processes, you can run other IDRISI modules using either **RunModule** or **RunProcess**. RunProcess is the same as RunModule except that it will autodisplay the resulting image or vector file, and requires that you give it display information.

After you launch a module with RunModule or RunProcess, you can inquire about the progress of that module using **GetProgress**. With this function you could run IDRISI in the background and report the progress of its modules in your program.

6. A VBA Example: Macro Scripting

The following example illustrates the use of the IDRISI API for macro scripting. It presents a simple illustration of what are known as a Cellular Automata (CA). A Cellular Automaton is an *agent* or *object* that has the ability to change its state based upon the application of a rule that relates the new state to its previous state and those of its neighbors. Thus the main elements of CA applications include cells, states, neighborhoods and rules. In this illustration of the Conway's "Game of Life", cells are the cells of a raster image, the states are either “dead” or “alive” (0 or 1), the neighborhood consists of the 8 neighbors bordering each cell (known as the “Moore Neighborhood” in the jargon of CA), and the rule is as follows:

Cells remain/become alive if within their Moore Neighborhood:

- The cell is dead and there are three living neighbors
- The cell is alive and there are 2 or 3 living neighbors

In all other cases, the cell dies or remains dead.

As is typical of most CA rules this logic can be implemented through iterative calls to the FILTER and RECLASS modules. In this specific case, the FILTER and RECLASS modules are used to determine the number of neighbors as follows:

- A user-defined 3 x 3 FILTER kernel with 10 in the center cell and 1 in all other cells is applied. Since the input consists of boolean data (0/1 for dead/alive respectively), the result will consist of cells with values between 1 and 8 if the center cell is dead and 11 and 18 if the center cell is alive.
- RECLASS then establishes the next state as follows. Totals of 3, 12 of 13 result in the cell being alive in the next state. All other values result in it being dead.

In order to run this script, two special files are required. The first is LIFE.RCL – a RECLASS file that specifies the rule just described. The second is an image of the initial state of cells called START (START.RST and START.RDC). All three of these files are included with this library in the self-extracting archive named “VBA_Example.EXE”.

To use the following code, place the files previously mentioned (LIFE.RCL, START.RST and START.RDC) into an accessible folder. This can be done by running the VBA_Example archive executable (from the Start Menu Run … facility in Windows) with the name of the target folder to which these data files should be extracted as a command line parameter (e.g., “VBA_Example c:\data”). Then run IDRISI and set the working folder of the current project to the same folder. Then exit from IDRISI. This last step is not necessary, but will be used to illustrate what happens when IDRISI is not running when the API is invoked.

Now start a new VBA project with a single form. Then add a button and a text box. The text box (TextBox1) will be used to specify the name of the start image (using its simple name, without a path or extension). The button (CommandButton1) will be used to execute the script. Then double click onto the button to move to the CommandButton1_Click() subroutine. Then add the following code:

```
Private Sub CommandButton1_Click()
```

```
        Dim inname,outname,oldname,tmpname As String
        Dim i,success As Integer
        Dim prefix,suffix As String


        prefix = IDRISI32.GetWorkingDir
        oldname = Trim(TextBox1.Text)
        suffix = ".rst"

        oldname = prefix + oldname + suffix
        success = IDRISI32.DisplayFile(oldname, "qual16", 0, 1, 2, 2, 0, True, oldname)

        For i = 1 To 5
          inname = oldname
          tmpname = prefix + "tmpx" + suffix
          outname = prefix + "tmp" + Trim(Str(i)) + suffix
          success   =   IDRISI32.RunModule("filter",   inname   +   "*"   +   tmpname   +
        "*6*1*1*1*1*10*1*1*1*1*0", True, "", "", "", "", True)
          success  =  IDRISI32.RunModule("reclass",  "i*"  +  tmpname  +  "*"  +  outname  +
        "*3*"+prefix+"life.rcl", True, "", "", "", "", True)
          success = IDRISI32.DisplayFile(outname, "qual16", 0, 1, 2, 2, 0, True, outname)
          oldname = outname
        Next i

End Sub
```

In this code example, the number of iterations is set to 5 – this can be changed as you wish, but should be set to be very small until the program is debugged (i.e., change it to 1 to start). Finally, to test your program, run the application and click on the button. Notice that Windows will automatically launch IDRISI if it is not already running.

7. A Delphi Example: Integrating a New Module

This example module is called Batch_Macro – a utility that runs the specified macro on every layer designated by a specific file mask (e.g., *.rst). It is a separate executable that communicates with IDRISI through the API. For users who do not own Delphi 5, the primary Pascal source file (Batch_Macro32.pas) can be viewed with any text file editor (such as NOTEPAD). In addition, the executable version of this file (Batch_Macro.Exe) has been included with the code so that you can experiment with the installation and menu alteration procedure regardless of whether you use Delphi 5.

Installing the Sample Code

The Batch_Macro project code is contained in a self-extracting archive named Delphi_Example.Exe. This executable should be run (from the Start Menu Run … facility in Windows) with the name of the target folder to which the project should be extracted as a command line parameter (e.g., "Delphi_Example c:\batch_macro_code"). Note that this archive also contains a copy of the final executable (Batch_Macro.Exe) and a test macro (test.iml). If you do not intend on compiling the executable yourself, you should place the executable into the IDRISI Kilimanjaro/MODS folder. The test.iml file should be placed in a working folder containing other IDRISI raster images. The test macro simply makes a call to SCALAR to add the value 1 to all cells in the image. These results are stored in new images based on a compound of the prefix and the old name.

Compiling the Sample Code

The Batch_Macro project assumes that you have already imported the IDRISI type library into Delphi (see Section 2). It will not compile until that has been achieved. The project is written in Delphi 5, and can be compiled into an EXE using Delphi's Build operation.

Operation of the Batch_Macro Module

This utility takes a standard IDRISI macro file and applies it to every file matching a specified wildcard mask in the designated folder. The macro can consist of a single line or it can contain multiple lines. All that is required is that there be at least two command line parameters – one for the input wildcard file mask (%1) and a second which specifies the prefix to use for all output files (%2). We suggest that for testing you use a wildcard file mask that will only result in a few files being processed.

Internal Processing Logic

The *real* work done by this module is all contained in the click event of the OK button. The general logic is as follows:

1. Construct a list of all files that match the wildcard mask.
2. Read the macro file and create a new macro file that contains the same code for each input file. Thus, for example, if the original macro contained 5 lines and 20 files matched the wildcard mask, the resulting macro file would contain 100 lines – all now reworked to indicate specific files.
3. Cycle through the new macro file using the API call to RunModule to execute each operation.

API-Related Calls Used by Batch_Macro

- The main form of this project is Batch_Macro32. At the start of its implementation section it contains a "Uses" statement to declare the IDRISI Type Library :

  ```
  uses IDRISI32_TLB;
  ```
- Then it declares a global variable in the VAR section of the implementation of Batch_Macro32 to be used as a reference for the interface of the IDRISI COM Server:

  ```
  IDRISI32 : IIdrisiAPIServer;
  ```

Note that this causes IDRISI to increase its client reference count by one.

- Next it establishes itself as a client of IDRISI in the FormCreate method of the main form (Batch_Macro32) as follows:

  ```
  if IDRISI32 = nil then  IDRISI32:=CoIdrisiAPIServer.Create;
  ```

  Note that at this point, Windows will load IDRISI if it is not already running.

- All remaining API-related calls are in the click event of the OK button. The first step is that Batch_Macro establishes itself as a registered process of IDRISI:

  ```
  ProcID:=IDRISI32.AllocateProcess;
  IDRISI32.Set_Process_ModuleName(ProcID,'Batch_Macro');
  ```

  Once this has been done, IDRISI is prepared to track the progress of the module and to receive error messages that it will report.

- As it processes macro calls, it sends progress report information and calls the appropriate IDRISI module using the RunModule method:

  ```
  IDRISI32.PassInProgress(ProcID,Kounter,Lines);
  IDRISI32.RunModule(ModName,Cmdln,True,'','','','',True);
  ```

- After the processing of the macro is complete, it signals that it has completed and then unregisters itself as an IDRISI process:

  ```
  IDRISI32.ProcessFinished(ProcID);
  IDRISI32.FreeProcess(ProcID);
  ```

- Finally, it removes itself as a client of IDRISI:

  ```
  IDRISI32:=nil;
  ```

  Note that this causes IDRISI to decrement its client reference count by one.

Integrating Batch_Macro with IDRISI and its Menu

The last step is to integrate your module into the IDRISI system. If IDRISI is currently running, close it and then complete the following steps:

1. Compile Batch_Macro and place the executable into the IDRISI Kilimanjaro\MODS folder.
2. Go to the IDRISI Kilimanjaro\EXTENSIONS folder and make a back up of the SYSTEM.IMN file (call it "SYSTEM_IMN.BAK").
3. Now open the SYSTEM.IMN menu script for editing (using NOTEPAD or any other text editor). Find the line in the Action List that reads:

   ```
   SYSTEM_MACRO_RUNMACRO=IDRISI32,MACRO_RunMacro
   ```

   And then add a new line right after it that reads:

   ```
   SYSTEM_MACRO_BATCHMACRO=BATCH_MACRO.EXE
   ```
4. Then find the following sequence in the menu specification:

   ```
   ITEMNAME SYSMenu_MacroRunMacro
   ```

```
        ITEMCAPTION Run Macro
        ACTION SYSTEM_MACRO_RUNMACRO
        HELPCONTEXT 0
```

Then add right after it the following sequence:

```
        ITEMNAME SYSMenu_MacroBatchMacro
        ITEMCAPTION Batch Macro
        ACTION SYSTEM_MACRO_BATCHMACRO
        HELPCONTEXT 0
```

Your new module is now completely integrated into the IDRISI system. Launch IDRISI and you will notice the Batch Macro entry in the file menu. Click onto the menu entry, and it should launch your module.

Other Important Notes

- The FormStyle property of the Batch_Macro32 form has been set to "fsStayOnTop". Since the window for this module is not a true child of IDRISI, this ensures that it will remain visible without disappearing behind IDRISI when the focus is changed. However, note that there is a potential problem with this logic – if other modules are running while your dialog is present and an error occurs, the error message will be hidden by your dialog. Another approach is to use a modal form.
- Note that the Batch_Macro utility is only intended as an illustration. Although it works as described, it does not contain all of the error handling characteristics of a normal IDRISI module.
- Clearly the derived macro could have been run with the RunMacro module. However, it was implemented using iterative calls to RunModule for the sole purpose of providing progress reports (which RunMacro intentionally does not do). The PassInProgress method was used, with the processing of each input file constituting a pass. The progress report bar is then controlled by the module(s) called by each macro line.
- Developers: there are no restrictions on your use of the API, so long as any end-user is in possession of a valid license of IDRISI. However, if your installation procedure involves the replacement of an existing menu script file, we suggest that you provide an automatic backup facility and instructions on how to install your addition manually.

10. A Python Example

Interactive Mode Example

```
Python 2.2.2 (#37, Oct 14 2002, 17:02:34) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> import win32com.client
>>> IDRISI32 = win32com.client.Dispatch('IDRISI32.IdrisiAPIServer')
>>> IDRISI32.DisplayFile('sierra1.rst', 'greyscale')
```

```
0
>>> IDRISI32.DisplayFile('sierra345', '')
0
>>> IDRISI32.RunModule('SCALAR', 'sierra1.rst*tmpout.rst*1*12', 1, '', '', '', '', 1)
1
>>> _
```

Note: Green font represents the command typed by the user and black font to represents the feedback from Python. The set of characters “>>>” are the Python prompt sign.

Some commands:

Activate the COM extension in Python:

```
import win32com.client
```

Activate the IDRISI API Server. IDRISI will launch if it is not yet running:

```
IDRISI32 = win32com.client.Dispatch('IDRISI32.IdrisiAPIServer')
```

Call the DisplayFile API Function with a grayscale raster image:

```
IDRISI32.DisplayFile('sierra1.rst', 'greyscale')
```

Call the DisplayFile API Function with a RGB24 raster image:

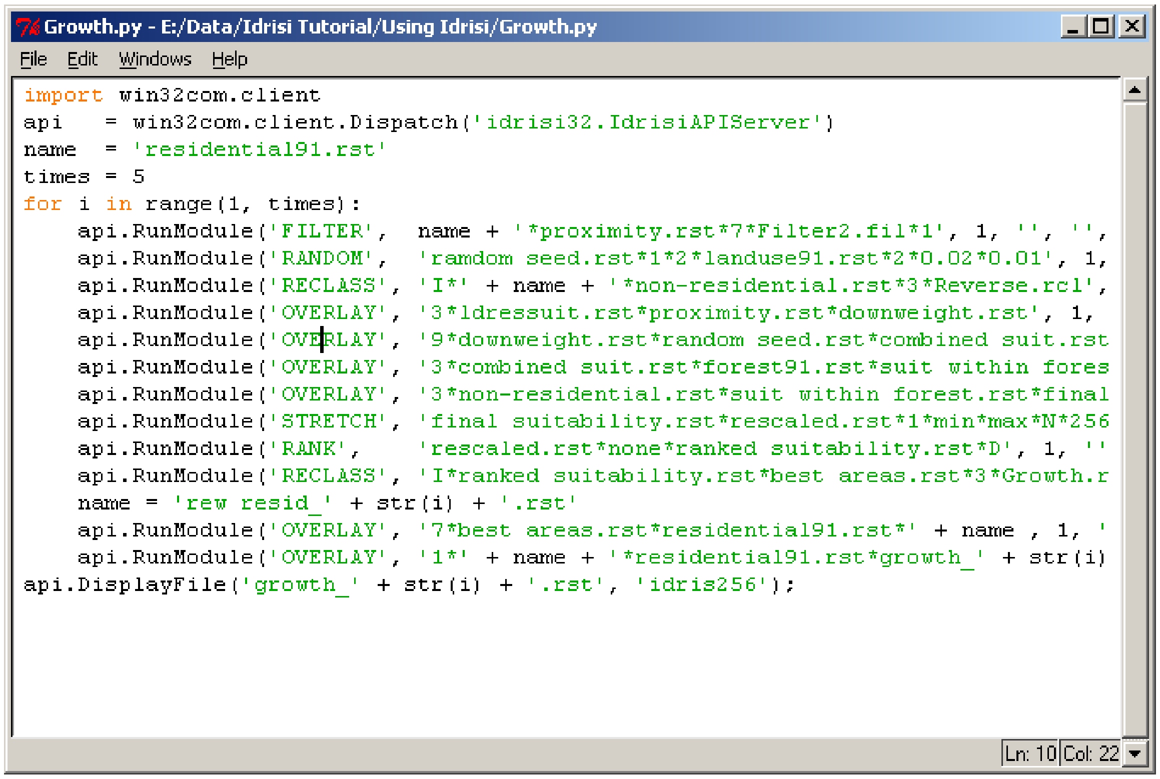
```
IDRISI32.DisplayFile('sierra345', '')
```

Call the RunModule API Function to run the SCALAR module:

```
IDRISI32.RunModule('SCALAR', 'sierra1.rst*tmpout.rst*1*12', 1, '', '', '', '', 1)
```

Programmatic Mode Example

The following graphic is an example of running the growth model found in the Introductory IDRISI Tutorial set using Python's programming language.



```
import win32com.client
api    = win32com.client.Dispatch('idrisi32.IdrisiAPIServer')
name   = 'residential91.rst'
times = 5
for i in range(1, times):
    api.RunModule('FILTER',  name + '*proximity.rst*7*Filter2.fil*1', 1, '', '',
    api.RunModule('RANDOM',  'ramdom seed.rst*1*2*landuse91.rst*2*0.02*0.01', 1,
    api.RunModule('RECLASS', 'I*' + name + '*non-residential.rst*3*Reverse.rcl',
    api.RunModule('OVERLAY', '3*ldressuit.rst*proximity.rst*downweight.rst', 1,
    api.RunModule('OVERLAY', '9*downweight.rst*random seed.rst*combined suit.rst
    api.RunModule('OVERLAY', '3*combined suit.rst*forest91.rst*suit within fores
    api.RunModule('OVERLAY', '3*non-residential.rst*suit within forest.rst*final
    api.RunModule('STRETCH', 'final suitability.rst*rescaled.rst*1*min*max*N*256
    api.RunModule('RANK',    'rescaled.rst*none*ranked suitability.rst*D', 1, ''
    api.RunModule('RECLASS', 'I*ranked suitability.rst*best areas.rst*3*Growth.r
    name = 'rew resid_' + str(i) + '.rst'
    api.RunModule('OVERLAY', '7*best areas.rst*residential91.rst*' + name , 1, '
    api.RunModule('OVERLAY', '1*' + name + '*residential91.rst*growth_' + str(i)
api.DisplayFile('growth_' + str(i) + '.rst', 'idris256');
```

To run Python will depend on which Python Interpreter is in use. From the main menu of "IDLE (Python GUI)": Edit/Run or press Control+F5. From the main menu of "PythonWin": File/Run or press Control+R. There is also a "Run" icon on the toolbar.

To modify the behavior of the example:

To change the number of iterations edit the integer value on line:

    times = 5

To display each intermediate result, add a tab in the last line. When the command "api.DisplayFile"is aligned to the lines above it, this command belongs to the statement block and it should run with every iteration.