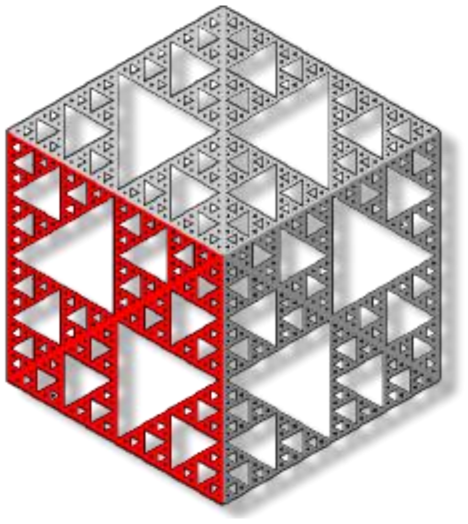


OMAX SCRIPTING

OVERVIEW:

OMAX Scripting allows someone familiar with computer programming concepts the power of writing scripts that access and leverage the internals of OMAX Applications. This allows one to extend and customize the OMAX software with new capabilities, while leveraging off many of the existing capabilities and commands already built into the OMAX software.

Reasons for scripting:

1. **Write your own commands or tools for OMAX LAYOUT** entirely using scripts. For example, make a tool for drawing star shapes, importing a specific type of file, generating an aerodynamic wing shape based on a formula, or digitizing points from MAKE for reverse engineering.
2. **Write custom events in LAYOUT or MAKE**, for example to modify the path as it is opened, send custom messages to the operator, or perhaps log some data, or even use scripts as custom report templates.
3. **Use scripts to interface with an external application** written in a different development system. For example, save out the CAD data, and then launch a program you wrote in C# to process it, save it back, and load back into LAYOUT.
4. **Write a custom file filter** to extend LAYOUT or MAKE's file importing capibilities to support new file types. For example, to be able to directly load and cut those part paths you already have for your laser.
5. **Write stand-alone applications** that interface directly with OMAX software. For example, one can write an application that gathers data off the network, generates special paths based on that data, reads some settings with a bar code scanner, and then launches MAKE.

This document provides a basic introduction into OMAX Scripting. It is assumed that the reader is already comfortable programming in a language such as Delphi, Object Pascal, C#, Visual Basic, C++, Java, etc. Though many non-programmers may find scripting to be a starting point to learn programming, if the web or other outside sources of "how to program" are consulted.

Warnings:

OMAX Scripting is **not** supported by OMAX Technical Support. For limited Technical Support, please email scripting@omax.com.

OMAX will from time to time make changes and additions to scripting that may cause previously written scripts to no longer function in future software updates. OMAX will, of course, attempt to minimize such changes.

Scripts and Plugins are executable code. **Be sure to only run scripts and Plugins from a trusted source**. As scripting gains popularity, so does the risk of someone posting malicious scripts for you to run. Use scripts with the same level of caution that you would use when installing 3rd party software. Antivirus software will NOT serve as protection against malicious scripts. If you do not trust the source, then do not run the script.

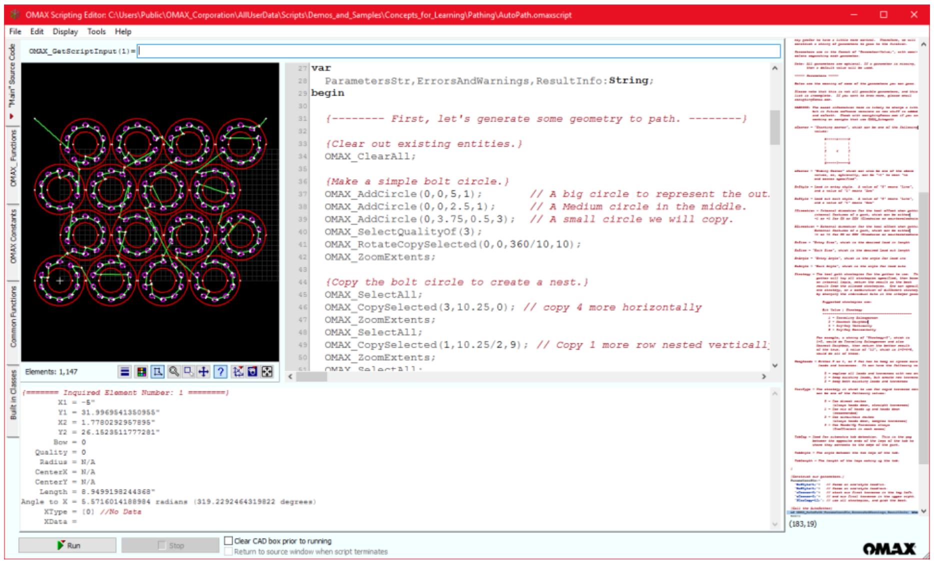
WHAT IS OMAX SCRIPTING?

OMAX Scripting is a complete but basic programming environment, compiler, and execution system built right into OMAX LAYOUT, MAKE and other OMAX applications. Its purpose is to allow 3rd parties to have an easy way to extend the OMAX software to do new things such as write new commands for OMAX LAYOUT, modify a path in MAKE, create an interface to a 3rd party plug-in, or even create new applications.

> **Warning**: With Scripting comes a lot of power. With such power comes the ability to make mistakes that can wipe out your data. Some of the script commands allow you to erase or overwrite files or folders. Be careful when using such commands, and be sure to back up your computer frequently.

Key point: OMAX Scripts always run inside the context of an OMAX CAD Box. In LAYOUT, what this means is that scripts run inside of the CAD box that is displaying the main editing window. In MAKE, what this means, is that scripts run inside the CAD box that is displaying the tool path in the main MAKE screen. Because of this, scripts generally work on CAD entities, tool paths, and other such local data items that are visible to the CAD box in which the script is running. (Though there are commands to access many external data sources such as files as well)

It is possible to run scripts as stand-alone applications. In this case, they essentially run in the OMAX script editor with all the editor tools hidden, giving the illusion that the script is essentially an executable. Running outside the context of LAYOUT or MAKE can allow the scripts, through advanced techniques, to automate entire systems such as opening a part in MAKE, processing the part, automatically start cutting, etc. (There are sample scripts that demonstrate this, such as Automated_Keychain_Maker.omaxscript.)


OMAX Scripting Editor: C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\Concepts_for_Learning\Pathing\AutoPath.omaxscript
File
Edit
Display
Tools
Help
"Main" Source Code
OMAX_Functions
OMAX Constants
Common Functions
Built in Classes
OMAX_GetScriptInput(1)=
Elements: 1,147
Run
Stop
Clear CAD box prior to running
Return to source window when script terminates
(183,19)

```
var
  ParametersStr,ErrorsAndWarnings,ResultInfo:String;
begin

  {-------- First, let's generate some geometry to path. --------}

  {Clear out existing entities.}
  OMAX_ClearAll;

  {Make a simple bolt circle.}
  OMAX_AddCircle(0,0,5,1);          // A big circle to represent the out.
  OMAX_AddCircle(0,0,2.5,1);        // A Medium circle in the middle.
  OMAX_AddCircle(0,3.75,0.5,3);     // A small circle we will copy.
  OMAX_SelectQualityOf(3);
  OMAX_RotateCopySelected(0,0,360/10,10);
  OMAX_ZoomExtents;

  {Copy the bolt circle to create a nest.}
  OMAX_SelectAll;
  OMAX_CopySelected(3,10.25,0); // copy 4 more horizontally
  OMAX_ZoomExtents;
  OMAX_SelectAll;
  OMAX_CopySelected(1,10.25/2,9); // Copy 1 more row nested verticall
  OMAX_ZoomExtents;
```

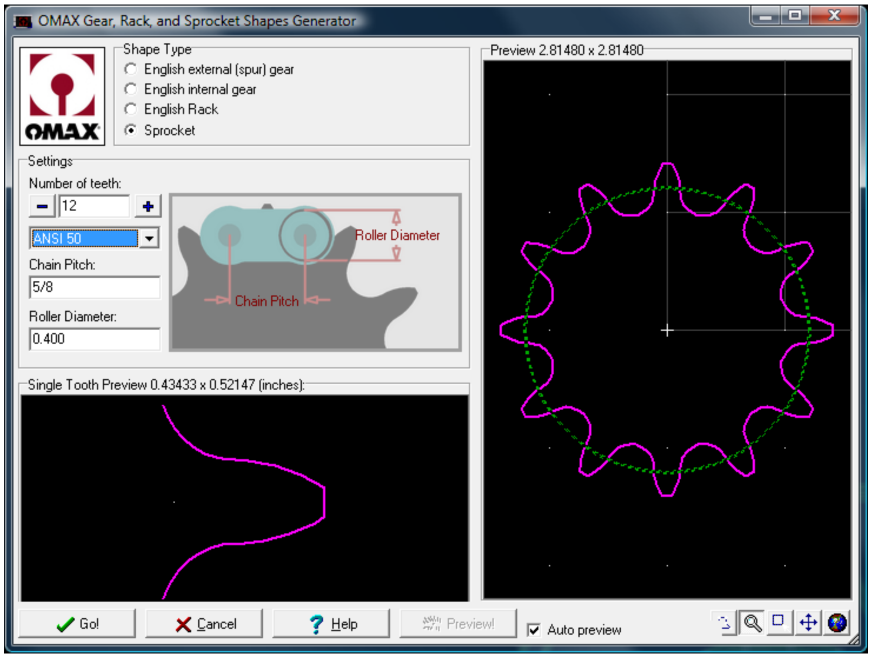
```
{======= Inquired Element Number: 1 ========}
        X1 = -5"
        Y1 = 31.9969541350955"
        X2 = 1.7780292957895"
        Y2 = 26.1523511777281"
       Bow = 0
   Quality = 0
    Radius = N/A
   CenterX = N/A
   CenterY = N/A
    Length = 8.9499198244368"
Angle to X = 5.5716014188984 radians (319.2292464319822 degrees)
     XType = [0] //No Data
     XData =
```

Above: The OMAX Script Editor

It is not necessary to do "CAD" work using scripting. While it is true that the CAD box is always the host for the script, the scripting engine is flexible enough that most any kind of computer program can be made using it. For an example of this, see CAD-Pong.omaxscript, which is a simple video game.

WHERE ARE SCRIPTS USED?

OMAX LAYOUT: Scripts can be used **inside of OMAX LAYOUT** to add new commands and functionality. For example, the OMAX Sprocket Maker, Move Lead, and Swap Lead commands were all done with scripts.



Shape Type
English external (spur) gear
English internal gear
English Rack
[✓ Sprocket]

Settings
Number of teeth: 12
ANSI 50
Chain Pitch: 5/8
Roller Diameter: 0.400

Single Tooth Preview 0.43433 x 0.52147 (inches):

Preview 2.81480 x 2.81480

[✓ Auto preview]

Above: OMAX Sprocket Maker in Intelli-MAX Premium, where the sprocket is generated with a script. (The user Interface in this case was not done with scripting, though in theory it could have been.)

Scripts can also be used to replace existing commands in LAYOUT, for example to replace the “line” command with one of your own making.

OMAX MAKE: Scripts can be added to OMAX MAKE, as “events” to modify tool paths prior to compiling, send messages to a user, and similar functions.

Stand alone utilities: It is also possible **to create stand-alone utilities** with OMAX Scripting that do not use LAYOUT directly, but leverage off of LAYOUTs geometry kernel, or other convenient Scripting features.

File filters: OMAX Scripts can be used to add support for additional file types in the File Open and Change Path Setup dialogs in LAYOUT and MAKE.

Note: OMAX Scripts will only run on computers that have the OMAX Intelli-MAX software installed.

Note: Some scripts will only execute if the software is officially registered.

FEATURES

OMAX Scripting is very extensive, and supports many powerful features including:

- Most capabilities of common programming languages, including variables, arrays, functions, procedures, loops, conditionals, etc.
- Built in functions for Math, Strings, Date/Time, etc.
- Multiple syntax support: Pascal like, Basic Like, and C Like syntaxes (**PascalScript strongly recommended;** the others syntaxes not as complete or as well supported / documented.)
- Direct access to many of LAYOUT's high-level geometry generation and processing commands
- Direct access to other OMAX functions, such as the QuickEval() math expression evaluator
- Large class library for the creation of user interfaces, bitmaps, database connections, and much more
- Lots of sample files to learn from
- Templates to use as starting points
- Basic script editor / debugger / previewer / executer

MISSING FEATURES

Although OMAX Scripting is very powerful, and reasonably complete, there are some features found in some modern development platforms that are not supported:

- No type declarations (records, classes) in the script (though many pre-defined objects and data types are exposed to scripting.)
- No pointers (though they are used indirectly in places, such as event handling in user interfaces.)
- No sets (but you can use 'IN' operator, for example: `a in ['a'..'c','d']`)

(Do not worry if you do not understand the above, as you do not need to in order to use OMAX Scripting.)

OTHER OPPORTUNITIES:

OMAX Scripting is just one of many ways one can customize or interface with the OMAX Software. **Be sure to check out the Developer's Guide in the regular help documentation** for more opportunities such as making your own Tool path Fonts, Custom Reporting in MAKE, Parametric Shape creation, writing Post Processors to interface between third party CAM systems and the OMAX, commanding MAKE to do automated tasks, monitoring MAKE's status with an application you wrote or a 3rd party application using MTConnect or Windows Messaging, and more.

Note: Developer options and available documentation and sample scripts vary depending on the brand, with the OMAX "Premium" software having all the options, MAXIEM having a few less, GlobalMAX fewer still, and soforth. For the most part, though, they are all pretty much the same. The OMAX Interactive Reference is the most complete documentation, and it can be downloaded from support.omax.com.

WHAT IS THE LEARNING CURVE LIKE? WHAT IS EASY, AND WHAT IS HARD?

For those already experienced with programming, expect that for the most part the language will be relatively easy to pick up, but more advanced projects may be more difficult, or even beyond the scope of scripting.

One can pick up the basics quickly. Study the sample files, play, and have fun.

"Beginner" things relatively easy to do:

- Opening and running existing scripts
- Basic manipulation of CAD data using built in functions. It is reasonably easy to add elments and move them around, in a manner similar to LAYOUT, by simply creating a series of commands.

"Intermediate" projects include:

- Editing existing scripts to tweak them to your needs
- Create very simple user interfaces with InputQuery and ShowMessage
- Programming up loops and conditionals to use programming logic and structure to generate geometry that is even more advanced.

Slightly more advanced:

- Opening code templates using "File / New from template" to do some common operations for which starter code is provided.
- Create event scripts for MAKE, plugins for LAYOUT, or file filter scripts for LAYOUT and/or MAKE to import custom file types.
- Using Scripting to act as a bridge to software written in another language or development platform. (Note: There is an included template for this. See the "plug-in_interface" template.)

Things more difficult:

- Writing scripts that require complex logic, knowledge of software architecture, advanced geometry and math.
- Advanced user interfaces with buttons, windows, and controls
  - Tip: Start with the sample files, and be sure to check out the samples here: C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\Concepts_for_Learning\+GUI_Advanced+
- Using some of the pre-built objects. Documentation for these may not exist. You may have to research similar components by searching for them on the Internet. In such cases, it may be better to use scripting as a way to communicate with software you write in a different system.

WHAT IS THE BEST WAY TO LEARN?

Assuming that you are already familiar with the basic concepts of programming (loops, variables, conditionals, and functions), and that you know the basics of the OMAX software (how to draw a part and such) you should be able to just dive right in. Start by loading sample scripts provided by OMAX, studying the code, running the scripts, and modifying them to suit your needs and experimentation.

Full Demos:

The …`\Scripts\Samples\`**`Demos`**`\` folder contains many sample scripts, some of them quite complex, for a variety of different purposes that you can adapt to your own needs.

Tips:

- In this document, the path to the "Scripts" folder, and other sample folders may be shortened to "…\Scripts\", but the full path is typically:

  `C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\`

- It is easy to find the demo scripts from within the OMAX Script Editor, by clicking on "File" and then "Open from Demos and Samples Folder…" from the main menu.

- There are some "**Cheat Sheets**" you may wish to open in the Script Editor, print out, and then keep by your side for reference. These are located in the following folder:

  …`Scripts\Demos_and_Samples\Concepts_for_Learning\+Fundamentals+\`

Simple examples of commands and concepts:

The …`\Scripts\Samples\`**`Command_Examples`**`\` folder contains very simple scripts illustrating singular concepts or commands. It is highly recommended that you study these scripts and read the comments.

Templates:

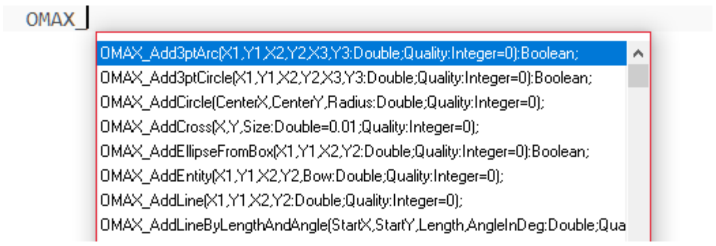
Once you know the fundamentals, a fast way to start making certain kinds of scripts is by starting from a template. To do this, from the script editor, choose "File", then "New from Template" from the main menu. A powerful one to start from is the template: 2_Click_Draw_Command.omaxscript, from which one can easily make modifications, and then save as a new script.

Important: Do not save your personal work in the same folders as the sample scripts, or other scripts provided by OMAX. **Save your work in the folders prefixed with "`User_`"**, not "`OMAX_`", or else you risk your work being overwritten or deleted by future OMAX Software updates.

REFERENCE MATERIALS EMBEDDED INTO THE SCRIPT EDITOR:

CTRL+Shift+Space to access common commands

Within the OMAX Script Editor, start typing in "OMAX_" then press Ctrl+SHIFT+Space on the keyboard. This will cause a list of all OMAX_ command (as well as some others) to appear:

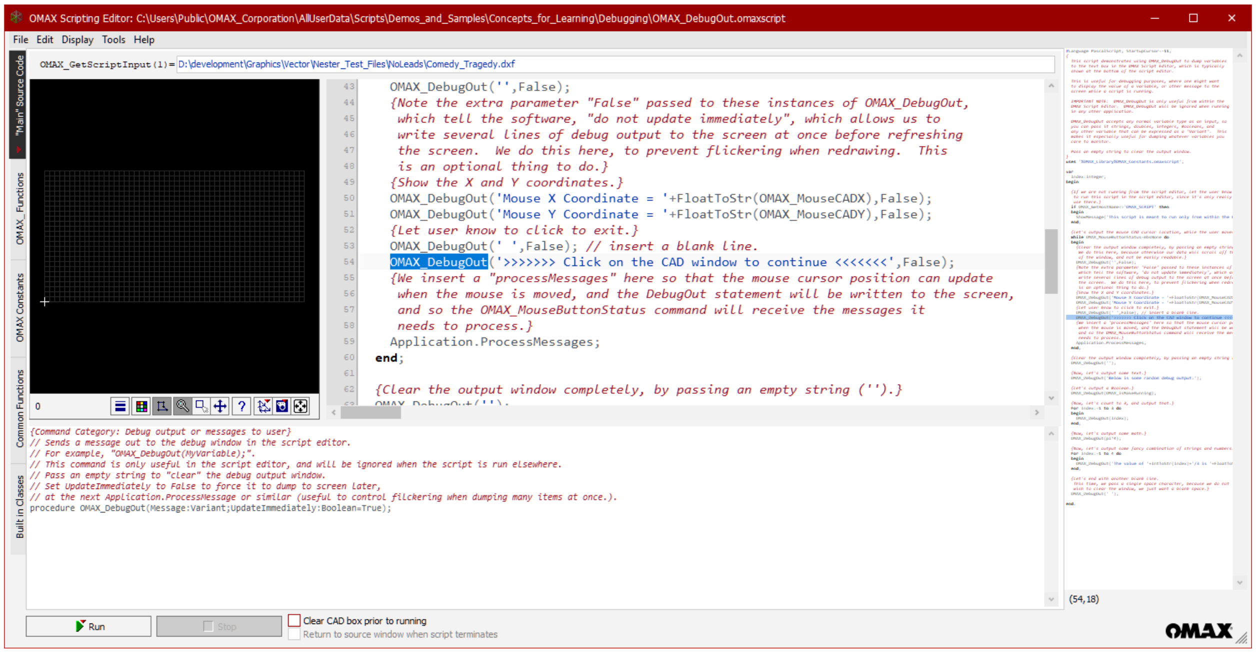


```
OMAX_
OMAX_Add3ptArc(X1,Y1,X2,Y2,X3,Y3:Double;Quality:Integer=0):Boolean;
OMAX_Add3ptCircle(X1,Y1,X2,Y2,X3,Y3:Double;Quality:Integer=0):Boolean;
OMAX_AddCircle(CenterX,CenterY,Radius:Double;Quality:Integer=0);
OMAX_AddCross(X,Y,Size:Double=0.01;Quality:Integer=0);
OMAX_AddEllipseFromBox(X1,Y1,X2,Y2:Double;Quality:Integer=0):Boolean;
OMAX_AddEntity(X1,Y1,X2,Y2,Bow:Double;Quality:Integer=0);
OMAX_AddLine(X1,Y1,X2,Y2:Double;Quality:Integer=0);
OMAX_AddLineByLengthAndAngle(StartX,StartY,Length,AngleInDeg:Double;Qua
```

Scroll through these, or continue typing, to find commands of interest, and press "Enter" on the keyboard to cause it to copy to the screen.

Highlight a command for context sensitive documentation

For any OMAX_ command on the screen, double click on it to highlight it. Any highlighted OMAX_ command will show up at the bottom of the screen with additional documentation:



OMAX Scripting Editor: C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\Concepts_for_Learning\Debugging\OMAX_DebugOut.omaxscript

OMAX_GetScriptInput(1)= D:\development\Graphics\Vector\Nester_Test_Files\NoLeads\Comedy_Tragedy.dxf

```
    OMAX_DebugOut('',False);
    {Note the extra parameter "False" passed to these instances of OMAX_DebugOut,
     which tell the software, "do not update immediately", which allows us to
     write several lines of debug output to the screen at once before refreshing
     the screen.  We do this here, to prevent flickering when redrawing.  This
     is an optional thing to do.}
    {Show the X and Y coordinates.}
    OMAX_DebugOut('Mouse X Coordinate = '+FloatToStr(OMAX_MouseCADX),False);
    OMAX_DebugOut('Mouse Y Coordinate = '+FloatToStr(OMAX_MouseCADY),False);
    {Let user know to click to exit.}
    OMAX_DebugOut(' ',False); // insert a blank line.
    OMAX_DebugOut('>>>>>>> Click on the CAD window to continue <<<<<<<',False);
    {We insert a "processMessages" here so that the mouse cursor position can update
     when the mouse is moved, and the DebugOut statement will be written to the screen,
     and so the OMAX_MouseButtonStatus command will receive the messages it
     needs to process.}
    Application.ProcessMessages;
  end;

  {Clear the output window completely, by passing an empty string ('').}
```

```
{Command Category: Debug output or messages to user}
// Sends a message out to the debug window in the script editor.
// For example, "OMAX_DebugOut(MyVariable);".
// This command is only useful in the script editor, and will be ignored when the script is run elsewhere.
// Pass an empty string to "clear" the debug output window.
// Set UpdateImmediately to False to force it to dump to screen later,
// at the next Application.ProcessMessage or similar (useful to control filckering when dumping many items at once.).
procedure OMAX_DebugOut(Message:Variant;UpdateImmediately:Boolean=True);
```

File Edit Display Tools Help
"Main" Source Code
OMAX_Functions
OMAX Constants
Common Functions
Built in Classes
Run
Stop
Clear CAD box prior to running
Return to source window when script terminates
(54,18)

Tabs with more documentation:

In addition, click on one of the tabs to the left side of the Script Editor for a full list of OMAX_ commands, and other supporting documentation.

WHERE CAN I FIND MORE HELP?

Reference material on the web:

Since the scripting engine is very similar to other languages, there are many resources on the internet to help you learn the basics of programming. The scripting engine also borrows heavily from the syntax and command structure of Embarcadero's Delphi Object Pascal language, in particular if you choose to use the Pascal Script variation of OMAX Script (the default and recommended configuration). Because of this, **you can often find command references by searching for "Delphi *CommandName*"** using your favorite search engine. Although many Object Pascal / Delphi commands are not supported by the script engine, many of the most common ones are, and the syntax is typically identical, though in some cases it may be slightly different.

Email based help:

We would like to hear from you if you have ideas or want to share what you are working on. We are unsure how much effort to spend on documenting and improving the script engine for you. If you are using it, we want to know! (Email: scripting@omax.com)

Please Remember: OMAX Scripting is not supported by OMAX Technical Support. This is a feature for advanced users who know something about programming already, or for internal use within OMAX to extend the OMAX software. If you have questions or comments, please feel free to email them to scripting@omax.com for "unofficial support", and we will be happy to help to the degree that we can.

OMAX CUSTOM SERVICES

OMAX offers fee based programming and hardware customization services. Projects can range from small, such as a custom tool path font, to giant, such as an automated workflow system. They can be pure software, purely hardware, or a mix of custom software and hardware. For more information on such services, talk to your sales person, or email omax@omax.com.

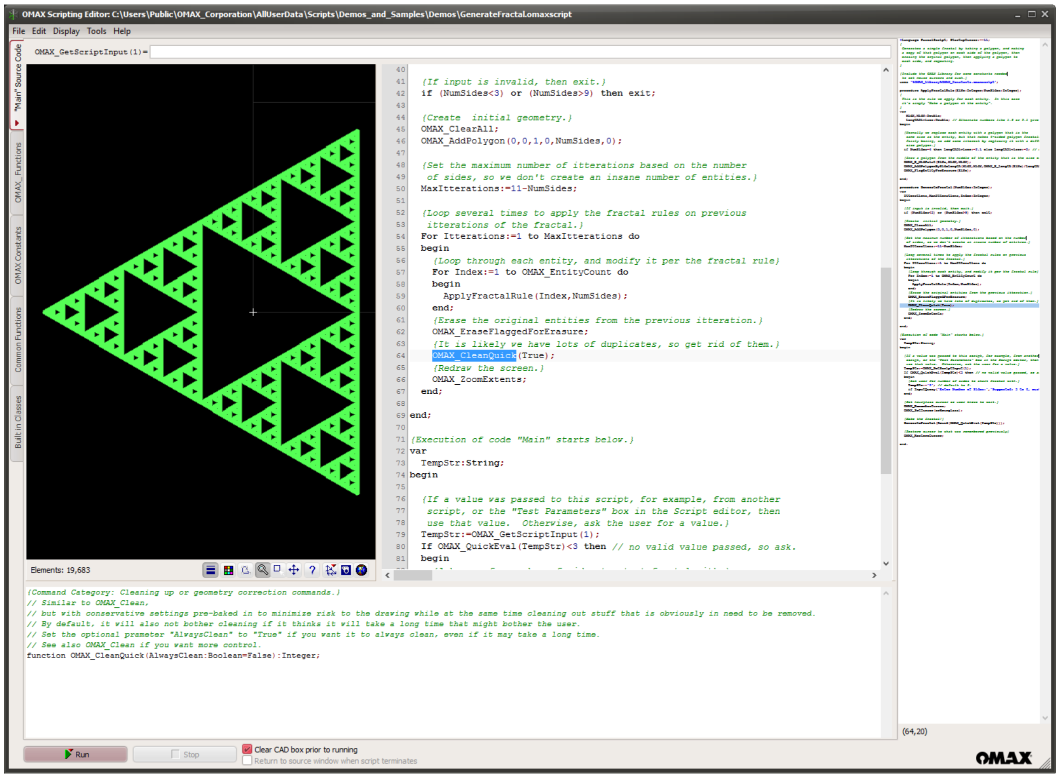
Examples of such custom programming services that we have done in the past for others include:

- Custom file filters to allow opening tool path files from a different machine directly into OMAX Make
- Several applications along the lines of "Grab a file from a database based on a bar code scan, tweak some settings, and launch MAKE to cut it"
- We have done a couple of robot interfaces to trigger a robot do do something at a certain event
- Custom camera vision system integration with AI based feature recognition.
- Custom tools in LAYOUT to automate away user-specific tasks.

Some of these projects have become plug-ins or sample files to share with you!

EDITING SCRIPTS

Included as part of the Intelli-MAX® software installation is a Script Editor (OMAX_ScriptEditor.exe) which is hidden away in the `Program Files\OMAX Corporation\OMAX_LAYOUT_and_MAKE\` folder. Although this is a convenient editor to use for script editing, it is not required that you use it. One can make OMAX scripts in nearly any text editor such as Notepad, Notepad++, Sublime, or other.


OMAX Scripting Editor: C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\Demos\GenerateFractal.omaxscript
File Edit Display Tools Help
OMAX_GetScriptInput(1)=
"Main" Source Code
OMAX_Functions
OMAX Constants
Common Functions
Built in Classes
Elements: 19,683
Run
Stop
Clear CAD box prior to running
Return to source window when script terminates
(64,20)
OMAX

```
  {If input is invalid, then exit.}
  if (NumSides<3) or (NumSides>9) then exit;

  {Create  initial geometry.}
  OMAX_ClearAll;
  OMAX_AddPolygon(0,0,1,0,NumSides,0);

  {Set the maximum number of itterations based on the number
   of sides, so we don't create an insane number of entities.}
  MaxItterations:=11-NumSides;

  {Loop several times to apply the fractal rules on previous
   itterations of the fractal.}
  For Itterations:=1 to MaxItterations do
  begin
    {Loop through each entity, and modify it per the fractal rule}
    For Index:=1 to OMAX_EntityCount do
    begin
      ApplyFractalRule(Index,NumSides);
    end;
    {Erase the original entities from the previous itteration.}
    OMAX_EraseFlaggedForErasure;
    {It is likely we have lots of duplicates, so get rid of them.}
    OMAX_CleanQuick(True);
    {Redraw the screen.}
    OMAX_ZoomExtents;
  end;

end;

{Execution of code "Main" starts below.}
var
  TempStr:String;
begin

  {If a value was passed to this script, for example, from another
   script, or the "Test Parameters" box in the Script editor, then
   use that value.  Otherwise, ask the user for a value.}
  TempStr:=OMAX_GetScriptInput(1);
  If OMAX_QuickEval(TempStr)<3 then // no valid value passed, so ask.
  begin
```

```
{Command Category: Cleaning up or geometry correction commands.}
// Similar to OMAX_Clean,
// but with conservative settings pre-baked in to minimize risk to the drawing while at the same time cleaning out stuff that is obviously in need to be removed.
// By default, it will also not bother cleaning if it thinks it will take a long time that might bother the user.
// Set the optional prameter "AlwaysClean" to "True" if you want it to always clean, even if it may take a long time.
// See also OMAX_Clean if you want more control.
function OMAX_CleanQuick(AlwaysClean:Boolean=False):Integer;
```

Above: OMAX Script Editor showing the source code editing tab.

The advantage of the OMAX Script Editor is that it supports syntax highlighting as well as some basic debugging tools, and has a built in "CAD Box" to conveniently test the scripts.

If one wishes to use a different editor for scripts, it is still possible to test and debug by dragging and dropping the saved .omaxscript files into a running copy of LAYOUT, or by other launching methods. (See *Opening, running, and installing scripts* below).

Tip: It is allowed to open more than one script editor at once. This can be handy when working with multiple source files, or to compare code, etc.

Tip: The script editor can also be handy for editing .ORD, .OMX and OMAX .DXF files, because one can edit the file, and click "Run" to preview them easily. This is particularly useful for OMX parametric shape editing.

OPENING, RUNNING, AND INSTALLING SCRIPTS

Scripts can be opened, edited, created, and run from within the OMAX Script editor. Scripts can also be opened and run in OMAX LAYOUT or OMAX MAKE, which of course is their ultimate destination in most cases.

OMAX Scripts always have the extension ".omaxscript" and the associated icon shown above. Internally they are just simple text files. One might take advantage of this unique file extension by telling Windows to associate this extension with LAYOUT, the Script Editor, or another editor of your choosing. (By default, .omaxscript files will open for editing in the OMAX Script Editor when double-clicked.)

To open a script in the script editor:

Run **OMAX_Script.exe** (Located in the `Program Files\OMAX Corporation\OMAX_LAYOUT_and_MAKE\` folder) and then choose "File / Open Script" from the menu.

The script editor has several tabs. The first is a CAD box similar to LAYOUT that can be used to view the results of the script, load / save DXF or other files to test scripts on, or check the results. The second is an editor for editing the script, and the remaining tabs are supporting documentation (presently quite crude, in the form of lists of commands and objects that you can use in your scripts. Although crude, it's a handy reference, and **can be searched with Ctrl+F**.) Also, the command list in the script editor may contain functions and procedures not mentioned in this document, so be sure to explore.

To open a script in OMAX LAYOUT:

To open from a menu: Choose "Scripts and Plugins", and then "Open and Run Script" from the main menu at the top of the screen.

Drag and drop / command line execution: One can drag and drop a script onto the desktop icon for LAYOUT, to have LAYOUT launch and run the desired script. Likewise one can launch a script via the command prompt or shortcut as in `LAYOUT.exe <ScriptName>`.

Drag and drop into LAYOUT Window: One can execute a script by simply dragging and dropping the script file into the main OMAX CAD window. This is a great way to test a script, or to auto-generate geometry by dragging and dropping the appropriate script from an explorer window.

Execute from a function key: See "Installing Scripts" below…

Replace an existing command, or trigger on an event: See "Event Scripts" further below…

Installing Scripts in LAYOUT to run via a hot key:

Scripts can be "installed" into LAYOUT for convenient access by assigning them to a function key. Doing this makes the script much more accessible and convenient. To assign a hot-key to run a script in LAYOUT:

1. Click on File / Configure Preferences from the main menu
2. Click on the "Hot Keys" tab
3. Configure a key to execute the "**Run Script**" command.

If no script is specified for the Run Script command, then when the hot key is pressed, it will ask you to open a script manually with a file/open dialog. Otherwise, specify a script to automatically open when the specified Function Key is pressed, by clicking on the button, and choosing the script to use.

Installing Scripts in LAYOUT as a "Plugin"

Sometimes scripts can be bundled together with other files into a single file called a "plugin".

Plugins are a way to bundle multiple files into a single group to "plug into" layout, in order to add new functionality. This makes it easy for the end user of the script to install it without having to have special knowledge of scripting, or having to hand-copy a bunch of files into folders. Plugins can be installed by any one of the following methods:

1. **Double click on an ".omaxplugin" file**. The .omaxplugin extension is registered to open in LAYOUT, so if you double-click it, LAYOUT will automatically run and install the script.
2. **Drag and drop an ".omaxplugin" file into LAYOUT**, or onto LAYOUT's desktop icon.
3. **Use the "Scripts and Plugins" menu** at the top of LAYOUT, and choose "Install a plugin…"

Once you have a script, making it into a plug-in is relatively easy (essentially, they are just .zip files).

Note: Plugins are described in more detail, including how to make them, later in this document, here.

Installing File Filter Scripts into LAYOUT or MAKE

Scripts that are named with the script name being a file extension (such as “cnc.omaxscript” for *.cnc files), and then saved into a special folder, can be used to expand the file opening capabilities of either LAYOUT or MAKE.

To add custom filters, place a script created as a filter into either the

“C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\User_Library\Filters\CAD\Import\”

..Or the …

“C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\User_Library\Filters\Path\Import\”

…folder. Files placed into the CAD folder will be recognized by LAYOUT. Files placed into the “Path” folder will be recognized by **both** LAYOUT and MAKE. LAYOUT and MAKE will automatically start using these in the “File/Open” and “Change Path Setup” dialogs.

Important: It is recommended that if you want **both** LAYOUT and MAKE to see your new filter, to put it into the **Path** folder. If you want **only LAYOUT** to see the new filter, put it in the “**CAD**” folder. **Do not** put your filter into both folders, or you will have duplicate entries for that filter in LAYOUT which will conflict with eachother.

More information on custom file filters, and how to make your own, can be found in the Creating Custom File Filters section of this document.

Installing “Event” Scripts into OMAX MAKE

To install an event script into OMAX MAKE, files are placed into the USER_Scripts folder, and must have a very specific name. MAKE will then automatically recognize them. Please see the special section on Event Scripts elsewhere in this document for details on this advanced topic.

Installing “Event Scripts” into OMAX LAYOUT

To install an event script into OMAX LAYOUT, files are placed into the USER_Scripts folder, and must have a very specific name. LAYOUT will then automatically recognize them. Please see the special section on Event Scripts elsewhere in this document for details on this advanced topic.

Using the Script Editor as a standalone executable:

The OMAX Script Editor can be launched from a batch file or Windows shortcut with the option to automatically start the script running and optionally to also hide all the developer tools. This allows one to make scripts that essentially run like a stand-alone Windows executable. One might use this feature to create a custom interface to launch MAKE – perhaps as an alternative to Parametric Shapes, or to grab instructions from the Network, or maybe a bar code scanner, etc.

Note: standalone projects still require the OMAX Intelli-MAX software to be installed in order to run.

The script editor (OMAX_Script.exe) can accept the following command line parameters.

- **<File Name>**
    - A file name of a script to run. (For example, "C:\My Files\MyScript.omaxscript"). Note that if the file name or folder path contains spaces, then it must be enclosed in double quote marks, because the command line is parsed using spaces.
- **AutoRun**
    - A switch to determine if the script should automatically start running.
- **NoUserInterface**
    - A switch to hide all the user interface elements relating to debugging and running scripts. If this switch is used, it is up to the script itself to create any necessary interface elements, though the CAD box will display.
- **HideAll**
    - Similar to NoUserInterface, except nothing is visible at all. In this case, it is up to the script writer to write all user interface elements, including creating a CAD box, if such a thing is desired. This is useful for stand-alone scripts where the CAD box is not used, but other user interface elements are created. If HideAll is enabled, then AutoRun will also occur automatically.

All of the above parameters can be passed in any order, and they are all optional. Generally speaking, one would always use "AutoRun" if "NoUserInterface" is enabled, since without the user interface, there is no way to run the script.

Note: When NoUserInterface is enabled, the script will not terminate automatically on most events that would normally terminate a script, such as pressing the Escape key. It is up to the script writer to provide an exit route for the user to terminate the script. The [X] button in the upper left of the window will still work to terminate the script and close the OMAX_Script.exe application.

Note: The words "AutoRun", "NoUserInterface", and "HideAll" can also be used as part of the folder path or filename of the script itself. For example, "C:\MyScripts\AutoRun\MyScriptNoUserInterface.omaxscript" will automatically run without a user interface when launched with the script editor by the command line. This makes it convenient to simply create a folder called "AutoRunNoUserInterface" full of scripts that one wishes to run in such a manner.

Additional parameters can be passed via the command line to control additional features of the Script executable:

Width=, Height= can be used to set the width and height of the window. Note: Window will appear centered in the primary display monitor, unless Top and Left are also specified.

Top=, left= can be used to set the top and left position of the window.

The QuickEval function evaluator (The evaluator used in OMAX Calculator and elsewhere) is used to evaluate the above, so one can say things like "Width=80*3", or pass any valid QuickEval information into it. In addition, two new items, "**Screen.Width"** and **"Screen.Height"** can be used to scale the window based on the screen resolution, so one can fill 80% of the screen width by saying, "width=Screen.Width*0.8"

Note / tip: Since this is using the QuickEval function evaluator, remember that per the rules of QuickEval, some functions you might use such as the "MAX" function require the use of a semi-colon (not a comma) to separate function parameters. For example, one might say Width=Max(Screen.Width*0.8;1000);

There are examples of a shortcut and a batch file, for starting a script:
(Shortcuts have a 260 character limit in the 'Target' field, so batch files are required for full use of these parameters.)

C:\...\Scripts\ Demos_and_Samples \Demos\Bow_Animation_shortcut.lnk

C:\...\Scripts\Demos_and_Samples\Demos\Bow_Animation_BatchFile.bat

See also: Distributing and sharing scripts

Other command line options for the OMAX Script Editor executable:

Encrypt: If the word "Encrypt" is passed to the command line, then the "Encrypt Script" function will automatically execute, just as if one had clicked on the "Encrypt Script…" menu item. Use this function with great caution, as there is no "decrypt" option!

BASIC SYNTAX AND STRUCTURE OF THE SCRIPTING SYSTEM

For those familiar with other languages, here is a quick breakdown of the basics of OMAX Scripting (which is based on the Pascal language / PascalScript):

- **Comments:** There are 3 types of comments supported to comment your code:

  ```
  // starts a comment for a single line of code

  {Curley braces can be used to comment out a large chunk of code that
  starts with the opening curly brace, and ends with the closing curly
  brace.}

  (* Parenthesis-Star works in a manner similar to curly braces, but
  is "more powerful" in that it can be used to surround {existing
  comments}, and is very useful to temporarily block out an entire
  chunk of code that may already contain comments.*)
  ```

- **Case insensitivity:** Unlike some languages, OMAX Scripting is not case sensitive.

- **Code blocks:** The "Begin" and "End" statements are used to define blocks of code, similar to the "{" and "}" symbols in some languages. For example:

  ```
  Procedure MyProcedure;
  Begin
        // Do something here
  End;
  ```

- **Declaring Variables:** The "**Var**" key word is used to declare a variable. Variables must be declared before they are used, and variables should be considered to contain random data unless initialized. The "var" key word is also used when passing parameters by reference in and out of functions. For example:

  ```
  Procedure GetAverageOfTwoNumbers(Num1,Num2:Double; var
  Answer:Double);
  Var
        Temp:Double; // declare a local variable.
  Begin
        Temp:=Num1+Num2; // Initialize it!
        {Pass back the answer, which was passed as
         a "var", so the new value will be passed
         back to the caller of this routine}
        Answer:=(Num1+Num2)/2;
  End;
  ```

Caution: Remember to always initialize your variables to a value before they are used! By default, OMAX Scripting (as well as Delphi, Object Pascal, and other similar Pascal derived languages), do NOT assign a value to the variable (say, "0" or "False", etc.) when the variable is created. Instead, the variable is created, and whatever happens to be in the memory space occupied by that variable becomes its value. While this provides greater performance for the code, it can also cause a lot of headache and voodoo bugs if you are unaware of this. See: VariableInitialization.omaxscript for an example demonstrating the potential consequences of not initializing your variables.

Caution / Tip: Variable names cannot be reserved words or names of other functions. For example, don't have a variable named "Min" because there is a function named "Min". The symptom of violating this rule may be difficult to diagnose. If you discover strange syntax errors such as "too many actual parameters" when attempting to call a function or use a variable, suspect that perhaps you have a variable of that same name declared somewhere.

- **Assigning values to variables:** The "**:=**" operator is used for assignment, while the "**=**" operator itself is used for evaluation of equality. For example:

```
Procedure MyProcedure;

Var
      X:Double;
Begin
      {Assign X a value of 3}
      X:=3;
      {Evaluate if X is equal to 3}
      If X=3 then ShowMessage('X is three!');
End;
```

- **Code lines are terminated with a semicolon**: Semicolons are generally not optional, and if left out, one can create a complex line of code spread out over several lines of text (for code reading clarity, for example), such as:

```
If MyVariable=3 then // first line
      MyOtherVariable:=30; // continuation of the same
```

- **Constants**: Constants are declared just like variables, except that they can be initialized to a value, and the key word "Const" is used instead of "Var". Note that constants cannot be changed once they have been assigned. For example:

```
Const PiOverTwo=3.14159/2;
```

- **Arrays**: Arrays can be simple to complex, static or dynamic, and one or many dimensions. For a complete discussion of arrays, search the Internet for “Delphi Arrays”. Otherwise, below is a very simple example of arrays:

```
Var

  {Declare Simple (Static) array of doubles}
  MyArray:Array[0..100] of Double;
  {Declare a Dynamic array of Integers}
  MyArray2 : Array of Integer;
  {Declare a 3 dimension array of type variant (which holds
   any other data type)
  MultiDimensionArray : Array[1..3,1..5,1..5] of Variant;  …

  {Show all values}
  For I:=0 to 100 do ShowMessage(IntToStr(MyArray[I]));
```

Notes:

Dynamic single dimension arrays can be resized using:

```
SetLength(ArrayName, size);
```

How a Static array is declared determines starting reference point.

```
o  For MyArray[1..3] //first entry is at MyArray[1].
o  For MyArray[0..3] //first entry is at MyArray[0].
```

See ..\Scripts\Samples\Demos\Arrays.omaxscript for more info

- **If / Then / Else:** The basic syntax for the "If" Statement is illustrated by the examples below:

  ```
  if X>20 then ShowMessage('X is bigger than 20!');

  if X<20 then
  begin
    X:=X+100;
    X:=X*3;
    ShowMessage('X was small, but I fixed it!');
  end;

  if X=3 then ShowMessage('X=3') else ShowMessage('X not 3');

  if X=7 then
  begin
    X:=5;
    ShowMessage('X was 7, now 5');
  end
  else
  begin
    ShowMessage('X was not 7');
  end;
  ```

- **The "Case" Statement:** The Case statement can be used to switch between various possible options based on an ordinal value. Note that the Case statement is not supported for strings, but one can of course use If / Else statements for that. Below is a simple example of the case statement:

  ```
  Case X of
    1: ShowMessage('X = 1');
    2..10: ShowMessage('X between 2 and 10');
    11: ShowMessage('X=11');
    else ShowMessage('X is weird.');
  end; // end of case
  ```

  See the "Missing Features" section above for information on how "Case" differs from the C "Switch" statement.

- **Strings:** Strings are declared as the “string” data type, can be easily assigned, manipulated and dissected with the various string handling commands. Below is an example of a string being declared, and added to another string.

```
var
  Greeting,FirstName,LastName:String;
begin

  FirstName:='Bob';
  LastName:='Jones';
  Greeting:='Hello there, '+FirstName+' '+LastName;

  ShowMessage(Greeting);

end.
```

- **Math Functions:** The script engine supports many math, trig, and similar functions and operators. The base operators are:

+	(`A:=B+C;`	add strings or numbers)
−	(`C:=B-A;`	subtract numbers)
*****	(`X:=2*Y;`	multiply numbers)
/	(`Y:=X/2;`	divide numbers)
div	(`I:=J div K;`	divide integers with integer result)

Note: To raise a number to a power, use the “Power” function, as in:

```
      Y:=Power(X,2); // "Y = X Squared"
```

Note: This is only a very small sampling of the scripting engines math power. The scripting engine contains many functions for simple and complex geometry, trig, and expression evaluation.

- **Boolean logic:** The script engine supports common Boolean operators:

AND, OR, NOT, and XOR

```
var
  A,B,C,D,E,F,G:Boolean; // declare some Boolean variables
begin

  A:=True;
  B:=not A;   // B is "False" now.
  C:=a or B; // if A or B is True, then C is true.

  if A=True then showmessage('A is True!');

  ShowMessage('The value of C is: '+OMAX_BoolToStr(C));

end.
```

- **Random**: The Random function generates a random floating point number between 0 and 1.0.   For example:

  ```
  ShowMessage(Random);
  ```

  …will show a random value somewhere between 0 and 1.

  ```
  ShowMessage(Random*100);
  ```

  … will show a random number somewhere between 0 and 100.

- **Uses**: It is possible to "use" or "include" additional source code files in your project. This allows breaking larger projects into smaller more manageable chunks (files) that can be maintained and reused easier. This is done with the "Uses" key word, which is very much similar to the "**Include**" key word in other languages. For example:

  ```
  {Include the OMAX Library for some constants needed
   to set mouse cursors and such.}

  uses '%OMAX_Library%OMAX_Constants.omaxscript';
  ```

  One can sort-of think of "uses" as meaning, "Cut and paste the following text file into right here."

  It is recommended to use the "User_Library" or "User_Source" folders to contain any secondary units to include. That avoids having to reference literal paths that may change depending on the end users setup. These library locations can be referenced by the %USER_Scripts% and %USER_Library% defined constants. For example:

  ```
  uses '%User_Library%MyLibrary.omaxscript';
  ```

  If you are unsure where these locations actually are on your machine, write a line of code to show them, like so:

  ```
  ShowMessage('%User_Library%');
  ```

  Note: It is possible to use libraries written in other OMAX Script syntax types. For example, one can "use" a unit done in PascalScript, even if the main unit is in the C syntax.

  Note: Basic and Jscript syntax types do not support "uses".

  For more information, see "Structuring Large Scripts"

- **The Application object:** The application object can be used to access properties of, or invoke methods of the main application. The main thing it is used for in OMAX Scripting is to return control back to LAYOUT (or MAKE, or other script enabled application), so that the application can process its messages (such as user mouse clicks, etc.) without appearing to the user as being locked up. For example:

  ```
  {Infinite loop waiting for SHIFT key to be pressed.
   Note we use Application.ProcessMessages to allow the
   dialog to receive mouse messages, and therefore allow
   the user to do things like stretch it, close, etc.}
  While OMAX_IsShiftKeyDown=False do Application.ProcessMessages;
  ```

BASIC USAGE OF THE OMAX SCRIPT EDITOR

The OMAX Script Editor consists of a CAD box for viewing the results of the script, a text editor for editing the scripts, and a page with a list of functions for convenience of cutting and pasting and searching.

Hot keys and their actions:

Ctrl+PgUp - Move to the begin of text
Ctrl+PgDn - Move to the end of text
Home - Move to the begin of line
End - Move to the end of line
Enter - Move to the next line
Delete - Delete symbol at right or selected text
Backspace - Delete symbol at left
Ctrl+Y - Delete current line
Ctrl+Z - Undo last change
Shift+C - Select the text block
Ctrl+A - Select all text
Ctrl+U - Unindent selected block
Ctrl+I - Indent selected block
Ctrl+C, Ctrl+Insert - Copy to clipboard
Ctrl+V, Shift+Insert - Paste from clipboard
Ctrl+X, Shift+Delete - Cut to clipboard
Ctrl+Shift+<NumberKey> - Set bookmark
Ctrl+<NumberKey> - Goto bookmark
Ctrl+F - Search text
F3 - Continue search
Ctrl+Shift+Space – Open context menu to browse many commands and functions

Important Tips:

- Start typing something, and **press Ctrl+Shift+Space to open a context sensitive drop down**, that will show many common available commands. Continue typing to refine the list, or use the arrow keys to explore.
- **Double click on any command that starts with "OMAX_"** to view additional help and information at the bottom of the script editor.
- Use **Ctrl+Mouse Wheel** to change the size of the text in the editor.
- Hold down the **SHIFT** key when choosing a color scheme from the drop down menu, to open an editor to **make your own color scheme**.

"HELLO WORLD" EXAMPLE SCRIPT (DRAWS A SHAPE WITH 'HELLO WORLD' TEXT)

The code is shown with comments explaining each line.

Code   Description / explanation

```
#Language PascalScript                     //This must be the very first line of the script file,
                                           //and tells the compiler, which syntax type to use
                                           //(PascalScript, BasicScript, C++script or JScript.)

Uses                                       //Include other scripts by listing them here.
'%OMAX_Library%OMAX_Constants.omaxscript';//'Uses' must be removed or commented out if not
                                           //needed.

Var                                        //Declare global variables
  FontFileName:String;                     //'Var' must be removed or commented out if not
                                           //needed.

Const                                      //Declare global constants
  MyMessage='Hello World';                 //'Const' must be removed or commented out if not
                                           //needed.

begin                                      //Pascal style requires this to indicate
                                           //'Start a section'

  OMAX_ClearAll;                           //Clear the screen

  OMAX_AddEntity(1,0,7,0,0,3);             //Add some entities
  OMAX_AddEntity(1,0,1,2,0,3);             //This is the syntax for AddEntity:
  OMAX_AddEntity(7,0,7,2,0,3);             //OMAX_AddEntity(X1,Y1,X2,Y2,Bow,Quality);
  OMAX_AddEntity(1,2,7,2,-0.3,3);

  FontFileName:=OMAX_GetSharedFolder+'Fonts\Stick.fnt';    //Build font path

                                           //Insert string
  OMAX_InsertPathText(False,1,1,0.5,'Hello World',FontFileName,True);

  OMAX_ZoomExtents;                        //Zoom so the drawing fills the screen

end.                                       //Pascal style requires closing a section. At the end
                                           //of the script, "end" is followed by a period. For
                                           //loops and other uses, end is followed by a semi
                                           //colon.
```

DEBUGGING

OMAX Scripting includes very basic debugging capabilities. Breakpoints and debug watches are not supported, but there are some alternatives…

OMAX_SetDebugMode

Calling OMAX_SetDebugMode(ShowDebugOutput:Boolean) with a parameter of True will open a debug window and allow you to step through the code. It does not display variable values or other information. That can be done by using the Instructions window (see **OMAX_ShowInstructions**).

ShowMessage

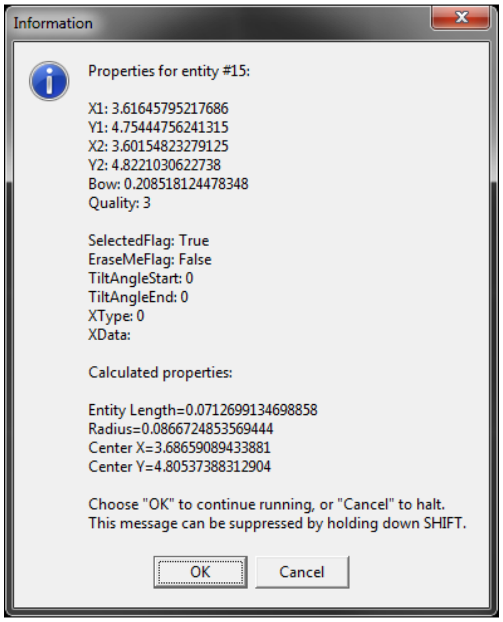
One can also use "ShowMessage" and other language features for debugging purposes**. In many cases, "ShowMessage" is a very convenient way to debug**. Search for ShowMessage elsewhere in this document for examples.

OMAX_ShowEntityProperties

There is a special command OMAX_ShowEntityProperties that does all the work of constructing a ShowMessage for you, that dumps the entity properties of a given entity number to the screen. For example:

```
OMAX_ShowEntityProperties(15);
```

…might show something like this as its output:



```
Properties for entity #15:

X1: 3.61645795217686
Y1: 4.75444756241315
X2: 3.60154823279125
Y2: 4.8221030622738
Bow: 0.208518124478348
Quality: 3

SelectedFlag: True
EraseMeFlag: False
TiltAngleStart: 0
TiltAngleEnd: 0
XType: 0
XData:

Calculated properties:

Entity Length=0.0712699134698858
Radius=0.0866724853569444
Center X=3.68659089433881
Center Y=4.80537388312904

Choose "OK" to continue running, or "Cancel" to halt.
This message can be suppressed by holding down SHIFT.
```

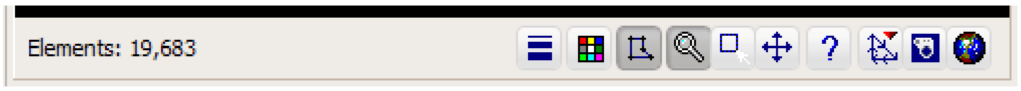
OMAX_DebugOut

Use OMAX_DebugOut to dump the value of a variable to the window at the bottom of the OMAX Script Editor. This command will only execute when running in the OMAX Script Editor. When running inside of another application such as LAYOUT or MAKE, OMAX_DebugOut will simply be ignored, and it is acceptable to leave this statement in your code for final release**.** **OMAX_DebugOut is perhaps the most convenient way to display the status of a running application**.

Tip: As with all OMAX_ script commands, double click on the command in the Script Editor for additional details and documentation.

CAD Tools in the Script Editor

The script editor also provides some tools for previewing the output below the CAD preview window:



The "Inquire" command on this toolbar (the "?" mark) is particularly useful for debugging.

Additional examples of debugging can be found as sample scripts here:

…`AllUserData\Scripts\Demos_and_Samples\Concepts_for_Learning\Debugging\`

DISPLAYING MESSAGES:

Displaying messages can be useful for both debugging and user interface. There are commands built into Windows for doing, this, which the script engine exposes. Below are some examples:

```
ShowMessage('This is a simple message with an OK button');

ShowMessage(5+3); // shows "8"

ShowMessage('This message...'+#13+'...is two lines long.');

ShowMessage('The current date and time is: '+DateTimeToStr(Now));

MessageDlg('This is a MessageDlg with an icon',mtInformation,mbOk,0);

If MessageDlg('Exit?',mtConfirmation,mbYes+mbNo,0)=mrYes then exit;
```

For more examples see: Scripts\Samples\Command_Examples\Misc_Commands\MessageDlg.omaxscript

If all that is desired is to simply show some simple text with an OK button, use ShowMessage. If more power is needed, including presenting a choice to the user, then use MessageDlg.

ShowMessage takes one parameter: A string, (though it can also take a variable or a number, as in the second example above.) Numbers and strings can be combined with the "+" symbol and a conversion function to convert the number to a string such as "FloatToStr" or "IntToStr", etc:

```
ShowMessage('Diameter = '+FloatToStr(2*Radius));
```

MessageDlg takes 4 parameters. The first is a string just like ShowMessage. The second is the kind of message, which determines what icon is displayed (Question mark, warning symbol, etc.) The third is an integer representing what buttons to show, and the forth is a help context ID which is not used and can be passed as zero. The result of the MessageDlg function returns which button was pressed.

These functions are mirrors of the Delphi equivalents, **except that for MessageDlg the buttons are passed by adding the button constants together with the "+" symbol instead of enclosing them in [brackets]** as one would do in Delphi. With that in mind, additional information on MessageDlg can be found here: http://www.delphibasics.co.uk/RTL.asp?Name=MessageDlg

SOME NOTES ON ANGULAR UNITS

There are two primary units of angle used in OMAX Scripting: Degrees and Radian. Generally speaking, unless specified otherwise:

- Functions and procedures that start with the prefix “OMAX_” expect values to be input in **Degrees**, and will output any angles in Degrees.
- Other functions (Sin, Cos, etc.) that are not prefixed by “OMAX_” typically use **Radian**

The function OMAX_DegToRad() will convert degrees into Radian. The function OMAX_RadToDeg() Will convert radians to degrees.

STRUCTURING LARGE OR COMPLEX SCRIPTS WITH “USES” OR “INCLUDES”

Typically there will be parts of code that will be re-used in many scripts. The approach of writing it once and then reusing the code through links (rather than copying and pasting) means you only have to change the code in one place if there needs to be a change. (The following examples are not complete working scripts, but just snippets to illustrate a point.)

Note 1: If calling a script from within a script, only the parent can use the predefined %variables%. When the compiler goes through and replaces the %variables% with the full string, it is only for the top level script.

Example:

MyScript.omaxscript has a statement "**Uses** '%USER_LIBRARY%\Common.omaxscript';"
In the unit “Common.omaxscript”, if you call another script, you could not use %USER_LIBRARY% or any other %variable%. You CAN call another script from Common.omaxscript, you just can't use a %variable%.

Note 2: Variables and Constants must be defined before they are used. This makes sense and is why you have to declare these at the top of the file. But what about variables shared between parent and child files? Uses statement are compiled before the variables for the parent script. So variables declared for the child scripts can be used in the parent, but variables declared in the parent file cannot be used in the child scripts.

Example:

Parent.omaxscript

```
#Language PascalScript
Uses
  ‘Child1.omaxscript’;
Var
  ParentStr:String;
begin
  ShowMessage(ChildStr);    //This will work
end;
```

Child1.omaxscript

```
# Language PascalScript
Var
  ChildStr:String;               // Note this is a global variable, not defined specifically for a procedure

begin
  ShowMessage(ParentStr);     //This will not work
end;
```

Note 3: Make sure each of the child scripts compiles by itself. Debugging can be difficult when you have more files connected.

Example A: In this scenario, you may get an error that Child1Procedure is not defined. This is correct but it is because Child1Procedure doesn't compile. So you will check your Uses statement and make sure the file path is correct, etc., but the problem is in Child1 itself, not in how you defined it.

Parent.omaxscript

```
#Language PascalScript
Uses
  'Child1.omaxscript';

begin
  Child1Procedure;
end;
```

Child1.omaxscript

```
# Language PascalScript
Procedure Child1Procedure;
begin
  OMAX_MISSPELLED_FUNCTION_CALL;
end;
```

Example B: An error message will show the incorrect information. The line number will be correct, but the text shown in the error message will be the text for the **parent** file, which may not be where the problem is.

Parent.omaxscript

```
#Language PascalScript
Uses
  'Child1.omaxscript';

begin
  Child1Procedure;
end;
```

Child1.omaxscript

```
# Language PascalScript
Procedure Child1Procedure;
begin
  ShowMessage(UNDEFINED_VARIABLE);
end;
```

The error message will state there is an undefined variable in line #4 and show the line text as ‘Child1Procedure’.

Again, you will check your spelling, etc. and not find anything. The error is in the **child** file, line #4.

SPECIAL %FOLDERLOCATIONS% TO CONSIDER

There are several folder locations where important files are stored, that your scripts may need to reference. Depending on the way the OMAX software was installed, and what operating system it is installed on, these folder locations may be in different spots. This makes it cumbersome to reference files in other locations, and make your script compatible with installations on other computers. Luckily, there is a fix, and the fix is easy using %FolderLocationVariables%.

As mentioned previously, before a script is compiled, all of the items below are searched for, and replaced with, the actual folder.

So, instead of saying

```
OMAX_OpenFile('C:\Users\Public\OMAX_Corporation\AllUserData\Samples\Art
_and_Fun\Honey.DXF');
```

…which will only work on some computers, say,

```
OMAX_OpenFile('%OMAX_SharedFolder%Samples\Art_and_Fun\Honey.DXF');
```

…which will open, regardless of the computer it is on.

Note: There is no need for a "\" character after the %. In fact, putting one there will cause the code to not function. Notice also the quotation mark right before the "%". The % path stuff is part of the string you are creating – not a function that is appended to it. Therefore it needs to be *inside* of the quotes.

The following lists the pre-defined % constants that can be used.

%OMAX_SharedFolder% this is the root of the "Public" folder that the OMAX software uses to store all of its sample files, scripts, shape library, and machine setup information. This location is public, because most of the data it contains needs to be shared regardless of which user is logged in to the computer.

%OMAX_Scripts% this is the root folder where all the OMAX scripts are kept (Scripts used for commands in LAYOUT or similar). **Do not write to this folder**, because data written there is subject to erasure or replacement in future software updates. Instead put your work in one of the folders prefixed by %User_

%OMAX_Library% this is the root folder that contains various units of code you might find handy to use in your projects. Again, **don't write here**, but you may want to include libraries that are in this folder in your code.

%USER_Scripts% this is the root folder where you can put your scripts

%USER_Library% this is the root folder where you can put your library items. For example, if you have a math library that is shared among several scripts, this is a great spot to put it.

Tip: If you can't remember the above, there is a shortcut in the OMAX Script Editor. Type the character "%", and then press **Ctrl+Shift+Space**. This will bring up code insight, with a list of keywords that start with "%"

```
//Examples using predefined folder references

#Language PascalScript

uses '%OMAX_Library%OMAX_Constants.omaxscript';

var

  Test_INI_File1:String;

const

  Test_INI_File2 = '%User_Scripts%' + ‘Test2.ini’;

begin

  Test_INI_File1 := '%User_Scripts%’ + ‘TestFile.ini';

  ShowMessage(Test_INI_File1);

  ShowMessage(Test_INI_File2);

end.
```

Tip: There are also numerous File and Folder related functions, which can be found in the OMAX_Functions tab, the Common Functions tab, and by pressing CTRL+SHIFT+SPACE in the script editor.

SCRIPT TERMINATION

In normal use, scripts will terminate under the following conditions:

- A new command is chosen by the user
- The user presses the Escape key
- The end of code is reached
- A critical error is encountered
- OMAX_ExitScript is called

If an abnormal termination is encountered (script was unable to finish), then the CAD box will be restored to the previous state it was in prior to running the script (kind of like the "undo" command was executed.)

There are times when one may wish to prevent the script from terminating – especially when working with objects, for example to build a user interface, or when working with files. In the case of working with objects, one might want to make sure the objects are freed before the script can terminate, or else they may hang around as zombie code. In the case of files, one might want to ensure writing to a file is 100% finished, and that the user cannot press "Escape" in the middle of it.

To force the script to ignore early termination by the user, use the "OMAX_AllowScriptToTerminate" command, and pass it a value of "False". Once the critical code has passed, pass it a "True" to allow control back to the user. For example:

```
{Do not allow script to terminate.}
OMAX_AllowScriptToTerminate(False);

  // Insert code that must not be interrupted here.

{Allow the script to terminate.}
OMAX_AllowScriptToTerminate(True);
```

Use this command sparingly. In most cases, it is very rare to need it. Always re-enable it as soon as you are done. Misuse of this command can cause the user to experience slowdowns and lock-ups of the user interface, resulting in an annoying experience.

Note: even with this command active, scripts may still terminate automatically in some cases, such as when an error is encountered, **OMAX_ExitScript** is called, if the "Stop" button is pressed in the Script Editor, or the script ends naturally.

Use the **OMAX_ExitScript** command to exit a script immediately. Note that if you created any objects, you should make sure they are freed before calling this (Or make sure they are assigned an "Owner" such as a UserForm, to take care of freeing the object for you.)

Note: OMAX_ExitScript takes an optional Boolean parameter to configure it to either "Exit as a Cancel" (the default), or exit normally. Choosing to exit as a cancel, will cause the script to terminate and restore the contents of the CAD box back to whatever it was prior to running the script. This is handy should the user of your script decide not to continue, and exit.   Setting this flag to “False” will cause the script to exit as though it had run to completion, so that whatever changes the script made to the CAD data will remain.

Note: Some commands will not allow script termination until the command is finished executing. This can cause your script to lock up, and possibly even lock up the script editor. Save your work often!

COLORS, STYLES AND THEMING

The OMAX software supports a feature known as "Styles" or "Theming". These styles/themes determine how text, user interface elements, and other graphics are rendered. Themese are controlled by a special file that overwrites whatever color or style element might have been defined by the programmer. Because of this, code that might set the color of an object such as `MyPanel.Color:=clRed`; may not change in color when a theme is active.

This is by design, because in most cases the theme should be in control of how things are displayed. There are special cases, however, where one may wish to override this behavior and force the control to be rendered as specified by the programmer, and not by the theme. This is done by passing the control into the function OMAX_SetStyleElementsForControl.

For example, one might have a user interface item such as the previously mentioned Panel, where it is desired that the panel be red. IN this case,the following code would be used:

```
MyPanel.Color:=clRed;
```

When running under OMAX software that is not themed, the above line of code will cause that panel to be red. However, if running on a themed application, that panel will be whatever color the theme has decide panels should be.

To ensure the panel is always red, regardless of any active themes, change the code as follows:

```
MyPanel.Color:=clRed;
OMAX_SetStyleElementsForControl(MyPanel);
```

This will have the affect of disabling all theming for "MyPanel", allowing the red to show regardless of whether theming is enabled or not.

There are 3 "Style Elements" that can be controlled:

```
seFont   =1; // Control the font
seClient=2; // Control the "Client" (background color)
seBorder=4; // Control the border style and color
```

To enable these items individually for a given control, the OMAX_SetStyleElementForControl function accepts optional parameters. Pass the items that you want the theme to control into this optional parameter, by adding them together. If you do not want the theme to modify something, then leave that item out. For example:

```
OMAX_SetStyleElementsForControl(MyPanel,seFont+seBorder);
```

…will cause the "font" and "border" styles to be handled by the active theme, if any, and the Client area will be whatever color you assign it.

For an example of using this function, see:

…\Concepts_for_Learning\User_Interface\DrawUserInterfaceOnTopOfCAD.omaxscript

CREATING CUSTOM FILE FILTERS WITH OMAX SCRIPTS

It is possible to extend the "Open" capabilities of LAYOUT and MAKE by writing a script to process a new file type.

To make a new file "Open" filter:

1. Create a new script
    a. **Tip**: There is a template for making file filters in the OMAX Script Editor under **File / New from Template**.
2. Save your script as follows:
    a. **The file name for the script should be the file extension you intend to support**. For example, if you want to create a file filter for *.cnc files, your file name would be "cnc.omaxscript"
    b. Save your script into either the "C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\User_Library\Filters\CAD\Import\" folder, or the "C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\User_Library\Filters\Path\Import\" folder
    c. **Note:** if you save in the "CAD" folder, then LAYOUT will recognize this new filter. If you save into the "Path" folder, then **both** LAYOUT and MAKE will recognize it. Do not put your filter into both folders at once, or they will conflict.
    d. **Tip:** There is a sample file importer for importing text (*.txt) files as tool path fonts located in the C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\Concepts_for_Learning\File_Filters\ folder. Using the instructions above, copy it into your Path\Import\ folder, then run MAKE, and open a Text file, and see how it works!
3. Write your script
    a. Use the following line of code inside of your script to get the name of the file to import:

    ```
    FileName:=OMAX_GetScriptInput(1);
    ```

    b. Create entities in your script however you like. Typically that would mean parsing through the input file, doing some conversions, and then using the OMAX_AddElement to generate the final converted data.
    c. If your file filter is for a tool path (saved in the "Path" folder for MAKE to use), then the elements must be in an ordered path sequence. If the file filter is intended for LAYOUT (saved in the CAD folder), then it is not necessary to maintain a continuous path sequence.
4. Notes:
    a. It is not necessary to do an OMAX_ClearAll at the start of the script, as that is for the host application to handle. In fact, by adding this line of code, it will break the ability to drag/drop into LAYOUT, and it will break the ability to use File/Insert.
    b. It is not necessary to do a redraw at the end of the importing, as that is redundant and handled by the host application. Adding redraw calls will simply slow things down for the user, though you may wish to have them temporariliy for debugging purposes.
    c. See also: Installing File Filter Scripts into LAYOUT or MAKE

DISTRIBUTING AND SHARING SCRIPTS

Our hope is that as scripting gains popularity, people will find new and unique ways to use scripting and share their work with others. Along those lines, we are starting to build in mechanisms by which one can more easily share scripting. Below are some options in this regard that are already available.

1. **Encrypt and sell:** It is possible to encrypt your scripts using the OMAX Script Editors encryption feature. Once encrypted, it is very difficult to decrypt the script into a human readable form, yet the computer will understand it fine. Below are some considerations when encrypting your script:
    a. Only bother to encrypt your script if you do not wish someone to be able to view its source code. This may be handy, for example, if you wish to write a script registration system in which someone must purchase your script for money, and you don't want them to so easily be able to hack it and use it without payment.
    b. Be extremely cautious to re-save the encrypted script to a new location and/or to a new name. If you overwrite your original source code with the encrypted form, then there is little hope in getting the original form back. Some of it can be recovered using special software at OMAX, but the recovered data will be in a form that is barely usable, as it will have lost a lot of comments, structure, and other human useful information.
    c. Only your "Main" unit can be encrypted. Included units imported with the "Include" or "uses" key words cannot be encrypted. Therefore, put the code you wish to encrypt into your main unit only.
    d. Consider using the function OMAX_GetUniqueComputerID in conjunction with code that you write for your encrypted script, to create password schemes that allow your script to only run on registered computers.
    e. Encryption is not perfect security, but we believe it to be enough protection to make it more worth it to buy your script than to hack it. However, OMAX makes no promises about this security measure. If the highest security is important to you, then you should make your application using something other than OMAX Scripting, so that you have full control over its security.
2. **Partner with OMAX:** If your script is of exceptional quality, and of use to other OMAX Customers, OMAX may wish to purchase it from you to distribute in future software updates. Contact scripting@omax.com to discuss. Note that our preference would be to share the source code, as that allows others to learn from it and/or make their own modifications. However we are open to discuss other arrangements. Please note that we cannot accept scripts from others that do not contain the source code for our review, because we must be able to verify the integrity of all of our source code in order to sell to certain high security customers (e.g. military).
3. **Wait for more options:** As time goes on, we hope to provide more options to make it easier to share scripts. These other options are in the brainstorm stage.

Caution:

Scripts and Plugins are executable code. Be sure to only run scripts and Plugins from a trusted source. As scripting gains popularity, so does the risk of someone posting malicious scripts for you to run. Use scripts with the same level of caution that you would use when installing 3rd party software. Antivirus software will NOT serve as protection against malicious scripts. If you do not trust the source, then don't run the script.

Contact scripting@omax.com for limited assistance for other possible options.

CREATING PLUGINS TO DISTRIBUTE YOUR SCRIPTS AND SUPPORTING FILES

Plugins are essentially .zip files, containing your script code and supporting files that can be "installed" into OMAX **LAYOUT**. When a plugin is "installed" it is automatically unzipped by the OMAX software, and copied into a folder in the "…Scripts\" folder.

Creating a plugin is appropriate if…

- The code is intended to be run in OMAX LAYOUT (only LAYOUT currently has plugin support.)
- You want to make it easy for the end user of your scripts to install them, or you wish to be able to quickly load and unload them from LAYOUT.

Plugins are not currently supported when generating stand-alone scripts or applications that are not intended to run inside of LAYOUT. For situations outside of LAYOUT, one must use the time-honored techniques of hand copying files, or creating a professional installer.

Plugins are .zip files renamed with the extension ".omaxplugin". When OMAX LAYOUT is installed, it registers the ".omaxplugin" extension with Windows, so that when a user double-clicks on a file with that extension, LAYOUT will automatically launch and install it.

By default, the name of the plugin in LAYOUT will be the same name as the .omaxplugin zip file.

TO MAKE A PLUGIN FROM A SCRIPT:

1. **Create a folder to contain all the files your script will need into that one folder**. Typically, this might be the folder in which you do all your development for that plugin.
2. **Name the folder** you created to be the intended name for your plugin.
3. **Create a "Main.omaxscript" file.** It is "Main.omaxscript" that will be called whenever LAYOUT runs the plugin. It must be named "Main.omaxscript", or the plugin will not run. Main.omaxscript must be placed into the folder you created.
4. **Use a zipping program to "zip" the entire folder into a single .zip file.**
    a. At this point, you should have a .zip file that contains a folder, which is named with the name of your plugin, and inside of that folder should be the file Main.omaxscript, along with any other files that your plugin might need.
5. **Name the .zip file to be the name of your plugin.**
6. **Re-Name your .zip file's extension,** with the extension ".omax**plugin**"

7. **Important**: Be sure to keep a backup copy of your files! The plugin, once installed, will be copied into the …Scripts\InstalledPlugins\ folder, but note that the folders in here will be **deleted**, when plugins are uninstalled. **Always keep a backup copy of your original work!**

TO TEST (OR INSTALL) A PLUGIN:

1. Double click on the .omaxplugin file. This will Launch OMAX LAYOUT, and automatically launch the plugin installer. (You can also install plugins by dragging and dropping them into LAYOUT, or using the "Install Plugin…" menu item in LAYOUT.)
2. Once the installer is done, you should see a new menu item in LAYOUTs, "Scripts and Plugins" menu at the top of the screen.
3. The installed files will be located in the …\Scripts\ folder, as a sub-folder with the same name as the plugin, but always prefixed with an underscore. For example, it might be "...\Scripts_MyPlugin\".
4. Click on your plugin from the menu, and test that it runs okay.

TO REMOVE A PLUGIN:

- Simply choose the "Remove plugin…" menu item in LAYOUT. (**Warning**: The plugin is deleted when you do this, unless it is backed up somewhere else.)
- Alternatively, you can delete your Plugins folder from the ...Scripts_MyPluginName\ folder. (In fact, all the "Remove Plugin…" feature does is send the plugin folder to the recycle bin.)

NOTES AND TIPS:

- **You cannot have two plugins of the same name.** If you attempt to install a plugin that has the same name as another, the new one will simply overwrite the old, and the old one will be gone completely.
- **You can install .omaxscript files, just like plugins:** If your script is only a single file that works all by itself then you can also install an .omax**script** file, using the "install plugin" menu item in LAYOUT. This can be a convenient way to add a simple script to the LAYOUT menu without bothering to make a full plugin.
  - **Note:** The installer will ask you for the name of your plugin. Note that it will choose a default name for the plugin based on the name of your .omaxscript file, but you can rename it to something else at this point, if you like.
  - **Note:** Not all **.**omaxscript files can be installed as plugins. Some .omaxscript may have dependencies on other files, or simply not be appropriate.
- **When plugins are installed, they are really just copied:** Behind the scenes, plugins are unzipped, and copied into the …\scripts_MyPlugIN\ folder. Each plug-in gets its own subfolder here, with the folder name being the name of your plugin, prefixed by an underscore character.
- **Uninstalling deletes the plugin folder:** When a plugin is uninstalled, the folder for it is sent to the recycle bin.
- **You can assign both scripts and plugins to function keys:** To launch a plugin from a function key, choose the "Assign Function Key" menu item in LAYOUT, choose the "RUN SCRIPT" command, and then browse to your "Main.omaxscript" file in the "…Scripts_MyPlugin" subfolder.
- **Plugins are for LAYOUT only.** Scripts for MAKE must be installed by hand, by copying the files across to the appropriate location, as described in Using Scripts in OMAX MAKE.

USING SCRIPTS IN OMAX MAKE

There are two main ways to run scripts from within MAKE:

1. As custom report templates
2. As "event scripts" in MAKE.

SCRIPTS AS CUSTOM REPORT TEMPLATES IN MAKE

In the **premium** version of OMAX MAKE only, it is possible to create custom report templates from nearly any text based file (.xml, .rtf, .txt, .html, .bat, etc.). Since .omaxscript files are also text based, the same concepts used for making other templates will also work for launching scripts.

Notes:

For more information on custom reports, please consult the OMAX Interactive Reference.

Custom reports are only available in the Premium version of OMAX MAKE.

For an example of such a report template, and additional documentation, please see: ScriptExample_Simple.omaxscript and ScriptExample_Advanced.omaxscript, which can be found tin the …`AllUserData\Reports\OMAX_Templates\` folder, and can be opened in the script editor to see the comments, or run as a report template from MAKE by choosing "File / Print / Pick Report Template…" from the pull down menu.

Special considerations:

- Scripts run as report templates from MAKE may not work well if they rely on relative paths to access external files. This is because the report generator makes a copy of the report in another folder, and then launches that file from that location.

EVENT SCRIPTS IN LAYOUT AND MAKE

It is possible to run scripts in OMAX LAYOUT or MAKE, for certain specific operations. This is done by creating "EventScripts" that are placed in the USER_Scripts folder, following a very specific naming convention.

The name of the script will tell OMAX MAKE or LAYOUT at what point in its execution it should open and run the script. For example, a script named `MAKE_OnApplicationStart.omaxscript` will run when MAKE first starts up, and `LAYOUT_OnApplicationStart.omaxscript` will run when LAYOUT first starts up. This might be useful to display a custom message to your operator.

Note that event scripts that start with the prefix LAYOUT_ will only run in LAYOUT, and those that start with the prefix of "MAKE_" will only run in MAKE.

Currently supported custom events:

For MAKE	
MAKE_OnApplicationStart.omaxscript	Runs when MAKE first starts.
MAKE_OnBeforeFileLoaded	Runs right before we open a new file to compile OMAX_GetScriptInput(1) will contain the file name of the file about to be loaded.
MAKE_OnAfterFileLoaded.omaxscript	Runs right after a successful open of a file to compile.
MAKE_OnBeforeApplyToolOffset.omaxscript	Runs right before the tool offset is applied OMAX_GetScriptInput(1) will return the Tool offset value entered by the user in MAKE as a Double. In the Premium version of the software, when Corner passing is enabled for Registered software, OMAX_GetScriptInput(2) will return a String, containing a list of corner pass lengths calculated for each Quality.
MAKE_OnAfterApplyToolOffset.omaxscript	Runs right after a successful tool offset application OMAX_GetScriptInput returns the same information as for MAKE_OnBeforeApplyToolOffset. * see important note below
MAKE_OnAfterCompileSuccess.omaxscript	Runs right after a successful compile has completed
MAKE_OnAfterCompileFail.omaxscript	Runs right after a compile has failed
MAKE_OnBeforePathDialogShow	Runs right before the "Begin Machining" path dialog shows.
MAKE_OnPathDialogShow	Runs right after the "Begin Machining" path dialog shows.

MAKE_OnAfterSetupDialogClose	Runs right after the Setup dialog (pump, timing, etc.) dialog closes.
MAKE_OnAfterAdminSetupDialogClose	Runs after the “Advanced / Administrator” setup dialog has closed.
MAKE_OnAfterPathDialogClose	Runs right after the “Begin Machining” path dialog closes.
MAKE_OnApplicationClose.omaxscript	Runs right before closing MAKE
MAKE_**Replace**ToolOffsetter.omaxscript	Runs **instead of** the normal tool offset routines. OMAX_GetScriptInput returns the same information as for MAKE_OnBeforeApplyToolOffset. * see important note below
MAKE_OnDryRunCompleted	Runs upon a completely finished dry run in the Path Control dialog (dry run reached the end of the tool path.)
MAKE_OnPathPaused	Runs when a path run in the Path Control dialog is paused by the user, fault, or other. Note: This script runs before messages (if any) are displayed to the user.
MAKE_OnDryRunPaused	Runs when a path that was dry-running in the Path Control dialog is paused by the user, fault, or other. This also runs prior to any error messages to the user.
MAKE_OnAllPathCyclesCompleted	Runs when the Path Control dialog has finished running all the cycles of the path, and has come to a complete stop. Note: If the user has entered a number of cycles to run for “repeat cutting”, this script will not run until they have all completed.
For LAYOUT	
LAYOUT_OnApplicationStart	Runs when LAYOUT first shows, before any files are loaded (note: When files load, they will override CAD data generated by this script.)
LAYOUT_OnAfterFileImported	Runs immediately after a file has been imported, but before it is cleaned, moved, zero length enties reomved, scaled, and other auto-fixing has occured. Useful to process files for custom importing needs, such as converting zero length entities into holes, converting entities on certain Layers into something else, etc.
LAYOUT_OnAfterFileImportFinished	Runs after file has completely finished importing and auto-fixes have been applied such as scaling, removal of zero length entities, fixing bows, and the like. Use this for general file processing of imported files, where you prefer the file to be in a cleaner state first.
LAYOUT_OnAfterFileOpened	Runs after a file has been opened via the regular “File/Open” command used to open OMAX files such as DXF, ORD, and OMX.

LAYOUT_OnAutoPathQuickCompleted	Runs after successfully applying a tool path using Autopath Quick (no dialog)
LAYOUT_OnAutoPathDialogCompleted	Runs after successfully applying a tool path using Autopath Advanced (the dialog)
LAYOUT_**Replace**GeneratePath	Use this to completely replace the "Post" / "Generate Path" command and associated dialogs. (The dialog normally used to generate .ORD and .OMX files.)
LAYOUT_**Replace**Command[***XX***]	Use this to replace most any LAYOUT command. The "XX" value in the file name must be the command number of the command to replace. For example, LAYOUT_ReplaceCommand[15].omaxscript will replace the Line command in LAYOUT completely. To determine what command number to use, select the command in LAYOUT's Function Key editing dialog, mouse over the command selected in the dialog, and notice the tool-tip / hint that shows the command number for the selected command.
LAYOUT_OnBeforeApplyToolOffset	Runs before applying the tool offset in the preview dialog for the tool path.
LAYOUT_ReplaceToolOffsetter	Runs instead of applying the tool offset in the preview dialog for the tool path – completely replacing it with tool offsetting commands that you write.
LAYOUT_OnAfterApplyToolOffset	Runs after applying the tool offset in the preview dialog for the tool path.
LAYOUT_OnApplicationClose	Runs as LAYOUT is closing, as almost the last thing that it does, after asking user to save if needed.

*** Note:** *The result of this script must be a continuous path with all entities of the original part "not selected", and all entities to convert to motor motions "selected". MAKE will then "Compile" the "Selected" entities into motor motions. This applies to MAKE_OnAfterApplyToolOffset.omaxscript, and MAKE_**Replace**ToolOffsetter.omaxscript*

To create a script that runs on any one of the above events, simply save your script into the **USER_Scripts** folder, with the name of the script being one of the event names listed above. In other words, if you wanted to show a message to the user, every time a file is opened, you might create a script that looks like this:

```
#Language PascalScript;
Begin
  ShowMessage ('Do not forget to turn on the water!');
end.
```

...and then it would be named, "`MAKE_OnApplicationStart.omaxscript`", and saved into the `C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\USER_Scripts` folder.

Now, whenever MAKE opens, the above message will show!

Important Notes about creating event scripts for LAYOUT and MAKE:

- These scripts run in the context of the CAD box, and manipulate the CAD entities. One can modify data prior to sending it into tool path interpolation, and one can send messages to users, open read files, generate logs, etc., but what is visible to the script is the CAD stuff. Scripting cannot be used to turn on the cutting head or modify an already compiled tool path (However, there are automation hooks that can be used to do this that are available within scripts, or by other programs you can write in other languages – see some of the scripting sample files, as well as the Developers Guide in the software documentation for more info.)
- If one needs to read some of the internal settings of MAKE, consider reading MAKEPreferences.INI or similar, using the scripts INI file commands. Note that the names of some INI files vary by brand.
- If one needs to write / modify some of the internal settings of MAKE, this is best done outside of MAKE, in a script running Stand-alone, or similar, when MAKE is not running. (For example, a script might set the material thickness, then run MAKE, wait for MAKE to close, then do something else. There are sample files that show this in the Script Sample File folder.) Likewise, modifying INI files used by LAYOUT, should in general be done while LAYOUT is not running.
- When modifying tool paths (or creating new paths in MAKE, etc.), it is very important that the resulting geometry be in a proper tool path sequence. Not all scripting commands preserve path sequence, so be careful about path sequence corruption.
- For some event scripts, depending on which of the above events the script is hooked into, one may have to set the "selectedflag" for each entity. In particular, the MAKE_ReplaceToolOffsetter and MAKE_OnAfterApplyToolOffset events will require that the SelectedFlag be set for any entity that the tool path interpolator is meant to process.
- It is possible to have more than one script attached to the same event by modifying how they are named. This is explained in detail later on in this document. (See "Multiple Scripts Attached to the Same Event")
- In many cases, it is quite easy for OMAX to insert and create additional events. If there is a particular moment you need a script to run, and it is not presently supported, please make a request to scripting@omax.com. It does not hurt to ask at least!

Tips:

- To temporarily disable a script, just rename it to something the script engine will not recognise. For example, to disable the `MAKE_OnApplicationStart.omaxscript` mentioned earlier, just rename it to something like "**XXX**`MAKE_OnApplicationStart.omaxscript`". This will cause MAKE not to find the file, so it will not execute. (Add the characters to the start of the filename, not the end. Adding them to the end will not work – see "Multiple scripts attached to the same event" below.)
- **For a sample script to get you started**, please see:

  ```
  C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\
  Concepts_for_Learning\EventScripts_MAKE\MAKE_OnAfterFileLoaded.omaxscri
  pt
  ```

  …of which you can copy into your

  ```
  C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\USER_Scripts\
  ```

  …folder, and then open the file

  ```
  C:\Users\Public\OMAX_Corporation\AllUserData\Scripts\Demos_and_Samples\
  Concepts_for_Learning\EventScripts_MAKE\InsertSerialNumbersHere.omx
  ```

  …to see it in action. Open it in the Script editor to study how it works.

MULTIPLE SCRIPTS ATTACHED TO THE SAME EVENT

Generally speaking, it is not necessary to write more than one script for a single event. Instead, one can simply create a larger single script to handle all tasks needed. However, there are cases where one might have more than one script attached to a single event. When that happens, MAKE follows specific rules on what order the scripts are executed.

Before going too far, it is important to note that there are two folders from which Event Scripts can be executed from. The first is the "User_Scripts" folder, which by now you should be familiar with, having read the sections above. The other folder is the OMAX_Event_Scripts folder, for which some scripts may have been placed there by the OMAX installer. Because there are two folders from which Event Scripts are executed from, it is possible for two scripts for the same event to be specified for the same event.

In addition, it is possible, even within the same folder, to have multiple scripts for the same event. This is done by adding additional characters to the script event file name. For example, to have two scripts that both fire when OMAX MAKE first starts up, one might place these two scripts into the User_Scripts foler, named as such:

```
MAKE_OnApplicationStart_A.omaxscript
MAKE_OnApplicationStart_B.omaxscript
```

In this case, when MAKE starts, it will see both of these scripts. The question then becomes what order will these appear in?

The default order in which a script is executed is alphabetical. In the above example, "`MAKE_OnApplicationStart_`**`A`**`.omaxscript`" would be executed before "`MAKE_OnApplicationStart_`**`B`**`.omaxscript`".

But what happens if the script name is exactly the same, such as the case where one script in the OMAX_Event_Scripts folder is the exact same name as the Event Script in the USER_Scripts folder? In this case, the OMAX script will go first. However, if one wishes to ensure that a User script will for sure execute before the OMAX one, then it is possible to add the key word "_First" to the file name. Likewise, if one wishes to ensure that the user script executes after all other scripts, then the key word "_Last" can be added to the file name. For example:

```
MAKE_OnApplicationStart_First.omaxscript
MAKE_OnApplicationStart_Last.omaxscript
```

It is completely possible to have several (or for that matter even hundreds) of scripts attached to a single event, and the order can be assured using the rules outlined above.

REPLACING OR DISABLING AN EXISTING OMAX_EVENT_SCRIPT

In the event that the OMAX installer has added an event script into the OMAX_Event_Script folder, and one wishes to disable or replace it, it is possible to do so with the **_Replace** keyword. For example, if the OMAX installer put the script "`MAKE_OnApplicationStart.omaxscript`" script into the OMAX_Event_Script folder, one can add a script into the User_Script folder named as such:

`MAKE_OnApplicationStart`**`_Replace.`**`omaxscript`

This script will now execute **instead of** the OMAX script. If this script is a valid script, but void of any content, it will simply disable the OMAX script for that event. Otherwise, it will execute whatever is in that script, instead of the OMAX Event Script of that type.

It is generally advised not to use this option, though it may make sense for some applications.

Note: Do not disable or replace scripts in the OMAX_Event_Scripts folder by simply deleting them, or editing them, because those scripts will be restored at the next software update, loosing whatever work was done in that regard. If one wanted to change the functionality of an OMAX script that exists in the OMAX_Event_Scripts folder, then the proper way to do so would be to copy that script over to the User_Scripts folder, and then rename it to have the _Replace keyword in the name. Doing so would help ensure that it is not overwritten in the next software update.

ALTERNATE METHOD FOR DISABLING A SCRIPT:

Us a *.ScriptSettings file prevent a script from running unless certain criteria are not met.

Presently, the only criteria supported is whether the CAD data in the CAD box that is running the script contains a certain kind of XData. (We use this internally to not bother running the Advanced Tool Offsetter in the Preview dialog for tool paths in LAYOUT if the tool path does not contain any XData commands to enable the Advanced Tool Offsetter.)

The solution is generic for any script: Create a file in the same folder as the script, with the same name as that script, but with the extension ".ScriptSettings". This is an INI file internally.

In that INI file, there is a section [RunOnlyIfThisXDataIsPresent]

If within that section, there is an item like this:

```
[RunOnlyIfThisXDataIsPresent]
XData_45=1
```

...then the script engine will interpret this as saying, *"If you don't see an XData 45 in the CAD data, then don't bother running the script."*

Change the number shown in the above example as "**45**" to the code number representing a different XData type, to suit your needs. (Find these code numbers listed in the drop down in the "XData Editor" dialog in LAYOUT.)

Add additional entries to have it check on additional XData items.

The Script Engine, at the very start of every script, will check for this file. If the file is not present, the script runs as normal. If the file does exist, then we look for the INI file entries and act accordingly.

Note: Presently, the only thing supported to look for is the presense of the specified XData. Over time, more criteria may be supported in this feature based on needs as they arrize.

This solves a couple of problems:

1. How to prevent a script from running if it does not need to?
2. How to know if the script needs to run or not, without actually loading and running it?
3. How to make scripts open and run as quickly as possible?

This avoids running a script unnecessarily; the software can run faster and be quicker for the end user.

STANDARD DATA TYPES, FUNCTIONS, OBJECTS, AND LANGUAGE ELEMENTS.

In addition to OMAX specific functions (which all start with the prefix “OMAX_”, and are described later in this document), there are many standard functions. Below are some of the most common ones to use:

Data types

Internally OMAX Scripting operates with the Variant type and is based on it. Nevertheless, the following predetermined types are available to use in your scripts:

Byte (Same as Integer type)
Word
Integer
Longint
Cardinal
TColor
Boolean (Boolean type “True” or “False”)
Real | Same as Extended type
Single
Double
Extended
TDate
TTime
TDateTime
Char (Character type)
String (String / text type)
Variant (Same as Variant type – holds just about anything.)
Pointer
Array

Not all of these types can be assign-compatible. Like in Object Pascal, you can't assign Extended or String to an Integer. Only one type - the Variant - can be assigned to all the types and can get value from any type.

THE OMAX CAD ENTITY ARRAY

The OMAX Scripting Language sits on top of the OMAX CAD Engine. As such, it can be used to access an array of CAD Entities. Typically, this array is accessed by one of the many functions that start with the prefix “OMAX_” such as:

```
function OMAX_E_X1(EntityNumber:Integer):Double;
```

…and…

```
procedure OMAX_E_SetX1(EntityNumber:Integer;NewX1:Double
```

These, and other similar functions, are used to get or set the X1 or other values of one of the entities in the array of CAD entities.

Each CAD entity is defined by the following attributes:

```
X1: Double;
Y1: Double;
X2: Double;
Y2: Double;
Bow: Double
Quality: Integer;
XType: Integer;
XData: String
SelectedFlag:Boolean;
EraseMeFlag:Boolean;
```

(There are other entity attributes also available for manipulation, but they are not discussed here, and are generally used outside the scope of what has been exposed to the scripting engine so far. For example, Z information, tilt information, and additional flags. For more information, email scripting@omax.com.)

How many entities are currently in the CAD system is defined by the “EntityCount” which can be accessed by the function “OMAX_EntityCount”. OMAX_EntityCount is a good function to call, for example, when looping through the array of all entities.

Notes:

- Many OMAX_ functions will directly modify the properties of entities. Sometimes it may modify them in a way you did not consider. For example, one can use the “EraseMeFlag” property of entities to flag an entity for future erasure, or perhaps some other flagging purpose. However, a call to a command such as “OMAX_FilletTwoEntities”, may internally use the EraseMeFlag when trimming the remainder of the filleted entities.
- The OMAX CAD Entity array starts with element #1 and ends with “OMAX_EntityCount”. If there are no entities in the drawing, then OMAX_EntityCount will be zero.

- When working on many entities in the array, be sure to look for an existing OMAX_ function that might already do what you need. Calling an OMAX_ function, instead of writing your own, will typically result in much faster performance.

CLASSES

You cannot define a new class inside OMAX Scripting, but you can use the external classes pre-defined within the scripting engine. For example, the TForm class can be used to create user interface elements, as in this example:

```
var
      f: TForm;
      b: TButton;

procedure ButtonClick(Sender: TButton);
begin
      ShowMessage(Sender.Name);
      f.ModalResult := mrOk;
end;

procedure ButtonMouseMove(Sender: TButton);
begin
      b.Caption := 'moved over';
end;

begin
      f := TForm.Create(nil);
      f.Caption := 'Test it!';
      f.BorderStyle := bsDialog;
      f.Position := poScreenCenter;
      b := TButton.Create(f);
      b.Name := 'Button1';
      b.Parent := f;
      b.SetBounds(10, 10, 75, 25);
      b.Caption := 'Test';
      b.OnClick := @ButtonClick;
      b.OnMouseMove := @ButtonMouseMove;
      f.ShowModal;
      f.Free;
end.
```

If you are familiar with Delphi, Lazarus, or similar, you can see there is little difference between PascalScript and Delphi code. You can access any property (simple, indexed or default) or method. All the object's published properties are accessible from the script by default. Public properties and methods need the implementation code - that's why you can access it partially (for example, you cannot access the TForm.Print method or TForm.Canvas property because they are not implemented).

Tip: One limitation of creating objects is that they are not automatically freed when the script is terminated unless they are "owned" by something that is. This can result in a form you created staying displayed on the screen after the script was killed by the user prematurely. For this reason, it is useful to use the pre-supplied "**UserForm**" objects for any form you create, as well as a parent and owner to attach your objects to, so that they are freed when the script terminates. These are described elsewhere in this document, and there are example scripts to reference in the Scripts folder. Generally speaking, when creating your own objects, if the object will accept an "owner", be sure to assign the owner to a UserForm. That way, when the script terminates, and when the UserForm is freed, your created objects will also be freed.

OMAX AND COMMON FUNCTIONS

There is a rich set of standard functions which can be used in an OMAX script. There are also OMAX specific CAD, geometry, and tool path related functions, which start with the prefix, "OMAX_".

For a list of what functions are available to you, click on the "**Common Functions**" tab in the OMAX Script Editor, or the "**OMAX_Functions**" tab. Another shortcut is to press **CTRL+SHIFT+SPACE** in the code editor of the OMAX Script Editor, and scroll through the list.

Documentation on all functions that start with the "OMAX_" prefix can be found by simply highlighting that function in the script editor, and observing the comments at the bottom of the script editor window.

For functions that do not start with the prefix "OMAX_", search for the function on the Internet, as these are common Pascal functions. For example, to learn about the "FloatToStr" function you might try "*Pascal* FloatToStr" or "*Delphi* FloatToStr" or "*Lazarus* FloatToStr", with "Pascal", "Delphi", and "Lazarus" being common languages that use Pascal as the base.

Note: For OMAX_ Geometry related commands that deal with length are typically in units of "Inches" unless indicated otherwise. Likewise, angles are measured in Degrees for OMAX_ Functions, while standard Pascal functions without the OMAX_ prefix are typically in radians.

EVENTS

You can use event handlers in the script. Unlike the Delphi event handler, script event handlers are not the methods of the object. The following example shows how to connect an event handler to the TButton.OnClick event:

```
var
      b: TButton;
      Form1: TForm;

procedure ButtonClick(Sender: TButton);
begin
      ShowMessage(Sender.Name);
end;

begin
      b := TButton.Create(Form1);
      b.Parent := Form1;
      b.OnClick:=@ButtonClick;// same as b.OnClick:='ButtonClick'
      b.OnClick := nil; // clear the event
end.
```

There are some predefined events available in FS_iEvents unit:

TfsNotifyEvent
TfsMouseEvent
TfsMouseMoveEvent
TfsKeyEvent
TfsKeyPressEvent
TfsCloseEvent
TfsCloseQueryEvent

TfsCanResizeEvent

ENUMERATIONS AND SETS

OMAX Scripting supports enumerations. You can write in a script: `Form1.BorderStyle:=bsDialog;`

Sets are not supported. However, you can use set constants in the following way:

```
Font.Style := fsBold; // Font.Style := [fsBold] in Delphi

Font.Style := fsBold + fsItalic; // Font.Style := [fsBold,
fsItalic]

Font.Style := 0; // Font.Style := []
```

ARRAYS

OMAX Scripting supports all kind of arrays: static (one- and multi-dimensional), dynamic, variant arrays.

For example:

```
var
    ar1: array[0..2] of Integer;
    ar2: array of Integer;
    ar3: Variant;
    SetLength(ar2, 3);
    ar3 := VarArrayCreate([0, 2], varInteger);
    ar1[0] := 1;
    ar2[0] := 1;
    ar3[0] := 1;
```

BUILT IN CLASS LIBRARY

Built into OMAX scripting is a very extensive class library that one can use to greatly expand a scripts capability to do just about anything including communication, bitmap graphics, user interfaces, forms, buttons, check-boxes, graphs, etc. Some of this is documented in the form of sample scripts you can learn from (in the Samples folders), while much of this presently remains undocumented.

To see what classes are available, click on the “Built in Classes” tab in the OMAX Script Editor, and explore the class tree. In many cases examples of how to use the classes can be found on the Internet by searching for “Delphi” followed by the class name, as in “Delphi TBitmap”. Although the classes in OMAX Script may not be exactly identical to the Delphi counterpart, they are often close enough that such an Internet search may be fruitful. Please note that although a particular class may be exposed for your use, not all of its methods will be exposed.

You may notice that all the classes start with a “T” for “Type”, which is a Delphi / Object Pascal convention.

SPECIAL USERFORM AND OMAXFORM PRE-CREATED FORM OBJECTS

UserForms solve problems that would otherwise occur if the standard TForm class is used to make user interfaces.

Scripts can terminate normally by running to the end of the supplied code, or they can be terminated prematurely by the user. If an object has been created within the script, and the script is terminated by the user or is otherwise terminated abruptly, then the objects created may not be freed.

For example, if one creates a TForm object that displays something to the user, and then the user presses "Escape" to terminate the script, the Form will remain on the screen as a zombie.

This can create memory leaks, visual clutter, and an annoyance to the user that is less than professional.

Another problem with using the standard TForm class is that it does not handle some standard OMAX behaviors. In particular, it does not pass the mouse zoom signals from the mouse wheel onto the CAD box.

The solution is to use one of the UserForm objects for each form you create. UserForm objects are pre-created by the scripting engine before your script even runs, and then they are freed automatically by the scripting engine after your script terminates. Therefore, if you use UserForm objects for your forms, you do not have to worry about them creating the pollution previously described.

It is considered "best practice" to use UserForm objects instead of manually creating TForms. In fact, we recommend that you never use TForm unless you have some special reason to do so.

Other objects can also have their "Parent" property assigned to a UserForm. By doing this, it ensures that the object will be automatically freed when its parent is freed. For example, if one creates a button on a form, the button will be freed when its parent form is freed. Even if your object is nonvisual, it is a good idea to assign its parent to a UserForm to ensure proper memory cleanup when the script is terminated.

There are 11 pre-defined UserForm objects available for use, numbered 0 to 10. They are **named with numbers** "UserForm**0**", "UserForm**1**", "UserForm**2**", etc. until "UserForm**9**". Since they are already created by the script engine, there is no need to create them yourself. In fact, **do not create UserForms yourself** – just use them.

Here is an example "clock" program that shows the time on a form.

```
{Set the form size to smaller.}
UserForm1.Width:=300; UserForm1.Height:=1;

{Show it.}
UserForm1.Show;

{Wait for user to close the form or press escape}
While UserForm1.Visible do
begin
  UserForm1.Caption:=DateTimeToStr(Now); // Show the time
  Application.ProcessMessages; // update screen
end;
```

Here are some additional notes regarding the code above:

1. We set the width and height of the form before we show the form. That ensures that the form knows its width and height ahead of time, so it can center on the screen properly. If we were to set the width or height after showing the form, then it would have started out centered, then grown and become un-centered. Tip: If you are attempting to set the width and height of the form to fit around some fixed objects like buttons on the form, consider using the “ClientWidth” and “ClientHeight” properties, since it’s otherwise awkward to deal with the fact that forms have a width to their border that varies based on different user preferences and versions of Windows.
2. The while loop ends when the form is no longer visible. This causes it to terminate should the user click on the “X” button on the form to close it.
3. The Application.ProcessMessages statement ensures that Windows messages (like to redraw the screen) happen properly. Without this, the clock would not update, and the user may think the program is locked.

Note: There is also a similar set of forms named “OMAXForm”. Do not use these. They are reserved for OMAX use to create forms (such as the snap toolbar) that you can use in your scripts. Use “UserForms” for your forms, and let OMAX use OMAXForms. This helps prevent you and OMAX from using the same forms at once.

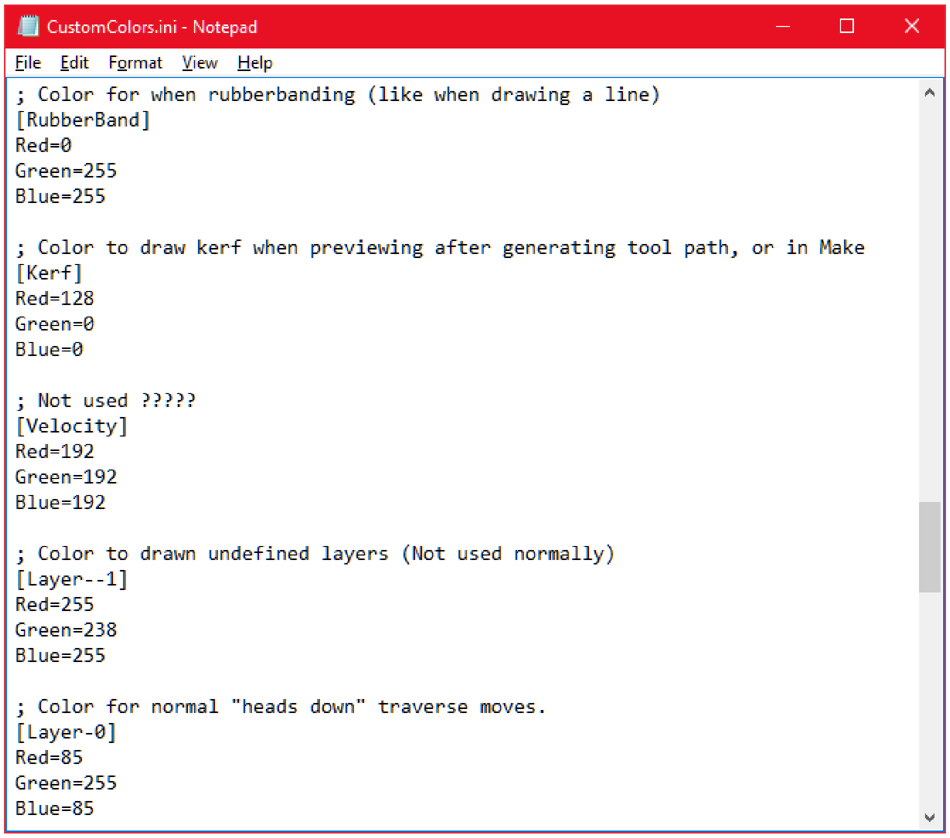
For a more in-depth example and additional notes, see:
…Samples\Command_Examples\User_Interface\UserForms.omaxscript

INI FILES: READING, WRITING, AND MODIFYING

The OMAX software uses INI files to store most of its internal settings and user preferences. It is possible to read and write to these files, as well as create your own INI files for your own storage needs.

In a nutshell, INI files are text files that specialize in storying configuration data. They consist of sections to organize the data, and then key / value pairs to store actual data items. They offer the advantage of being simple, robust, and easy for both humans and computers to read and modify.

Below is a screenshot of Colors.INI (a file used to hold the custom color preferences for OMAX LAYOUT), as seen when opened in Notepad:



```
; Color for when rubberbanding (like when drawing a line)
[RubberBand]
Red=0
Green=255
Blue=255

; Color to draw kerf when previewing after generating tool path, or in Make
[Kerf]
Red=128
Green=0
Blue=0

; Not used ?????
[Velocity]
Red=192
Green=192
Blue=192

; Color to drawn undefined layers (Not used normally)
[Layer--1]
Red=255
Green=238
Blue=255

; Color for normal "heads down" traverse moves.
[Layer-0]
Red=85
Green=255
Blue=85
```

Lines that start with a semi colon are comments that the computer ignores, but are sometimes found in INI files to make them more human-readable.

Items in [brackets] are sections, indicating that the items below it belong to a specific category of data, so as not to be confused by similar named values elsewhere in the file. For example, in the screen shot above, there is a section [Layer--1] with Red, Green Blue values below, and there is another section [Layer-0] that also has Red, Green and Blue values that can be assigned separately without any confusion, since they are in their own sections.

Key / value pairs are the items indicated by a key on the left, and equals sign, and the value assigned to that key on the right. The Red, Green, and Blue values shown in the example above.

There are many INI files used by the OMAX software, which you can find located in the C:\Users\Public\OMAX_Corporation\AllUserData\ folder. What files are there, and how or whether they are used will depend on the brand of the software (OMAX / MAXIEM / GlobalMAX, etc.), as well as what accessories are attached.

INI files are read from and written to using special functions in OMAX Scripting:

```
OMAX_ReadINIBool
OMAX_ReadINIFloat
OMAX_ReadINIString

OMAX_WriteINIBool
OMAX_WriteINIFloat
OMAX_WriteINIString
```

Notice that each function needs to know the INI file name to read or write, the Section to read or write, and the identifier (or key) value to read or write to.

For details on how these functions are used, please see the sample scripts located in the …Scripts\Demos_and_Samples\Concepts_for_Learning\Files_and_Folders\ folder.

Important Notes:

- The OMAX Software will use different INI files depending on brand for some applications. In particular the main preferences files for LAYOUT and MAKE are “Preferences.INI” and “MAKEPreferences.INI” when using the OMAX Premium software, but are “Preferences_B.INI” and “MAKEPreferences_B.INI” when running the MAXIEM Standard software. In order to know which one to talk to with your script, use the functions OMAX_GetMAKEPreferencesINI and OMAX_GetLAYOUTPreferencesINI, which will return the appropriate INI file name for whatever brand software is running.
- Inside of all OMAX Generated INI files, OMAX has standardized on Imperial units (Inches, Inches per minute, etc), and Degrees for all angular measurements.
- An INI file does not need to exist in order to be read. If the INI file is not found, the functions for reading the file will use a default value. Likewise, sections and Identifiers within the INI file also do not need to exist when reading. Again, the default value specified in the reading function will be used. This means that it is possible to “restore an application to defaults” by simply deleting the INI files. Do this with extreme caution, though, as the “default” values are unlikely to be what you want to configure your exact machine for. In fact deleting MAKES INI files will certainly corrupt the ability to run your machine.
- Boolean values are stored in INI files as the value of “1” to mean “True” and “0” to mean “False”.
- .INI files can be edited in nearly any text editor such as Notepad, Notepad++, or even the OMAX Script Editor.

Warnings:

- Always edit INI files with caution, since a minor change can vastly confuse or corrupt the software.
- It is best to read and write an applications .INI file when that application is closed. Otherwise, the application will likely overwrite your changes, or other collisions may occur.

MAKE'S MAKEPREFERENCES*.INI FILE

Here we document **some** of the entries in the INI file used by OMAX Make. Remember, to get the file name for this, use the function OMAX_GetMAKEPreferencesINI, to ensure you get the correct version for the brand of software that the script is running on.

```
; The materials SETTINGS section is used to configure settings
; related to how this path will be cut, but not the material itself.
[MaterialSettings]
NumWiggles=0
LowPressurePierce=0
LowPressureEtch=1
LowPressureScribe=1
LowPressureWaterCut=0
LowPressureCut=0
EtchSpeed=50
ScribeSpeed=50
WaterOnlySpeed=100
; The materials SETUP section is used to configure what
; material MAKE is configured to run for its next path.
[MaterialSetup]
MaterialName=Aluminum 6061
MACHINEABILITY=215.3
Pierceability=215.14
MaterialCategory=3
THICKNESS=0.5
ToolOffset=0.0135
ROTATION=0
Flip=0
SCALE=1
OffsetLeads=0
BrittleFlag=0
; Use the startup section to specify what file to open next time
; MAKE opens up.
[STARTUP]
LastFileLoaded=C:\MyFiles\MyArtwork.omx
```

This is only a tiny sample of the items available in MAKE's primary INI file. For additional help and information, open the INI file in a text editor and expore (with caution!). If you need any assistance with this file, or other OMAX INI files, email scripting@omax.com with specifics on exactly what is needed.

Note: What is available to edit in the INI files will depend on the brand of the software. Some INI file settings are hard-coded into the software and ignored. This is particularly true for all INI files that end with the suffix **_C.**INI. In these cases, modifying the INI file will have no effect.

INTERCEPTING AND PROCESSING WINDOWS MESSAGES SENT FROM MAKE

When OMAX MAKE is configured as a machine tool, and running, it is constantly broadcasting its status via Windows Messages. Other applications or services running on the same computer can then listen in on these messages. For example, one might monitor the machine, log the messages, or pass the messages on to somewhere else.

> The OMAX MTConnect Adaptor is one application that listens to the messages that come from make, re-formats them into the format expected by the MTConnect protocol, and then passes them over the Internet to 3rd party MTConnect applications. (For more information on this, search for "MTConnect" in the software documentation.)
>
> OMAX Intelli-VISOR also makes great use of the messages broadcast by MAKE, so that it knows what MAKE is doing, and can compare that with its sensors in order to know if anything is wrong.

Intercepting these messages is very easy using OMAX Scripting. Simply use the function:

```
function OMAX_GetRawMessageFromMAKE(IDNumber:Integer; var
Value:Integer):Boolean;
```

…which will return whichever of the many data items that OMAX MAKE sends that you request with IDNumber. The IDNumber value is the ID of the particular data item to return, and "Value" will return the value of that data item. For example, if one wanted to know if the machine is presently running a tool path, then the following code would do this:

```
var IsMAKERunning:Boolean;
begin
  OMAX_GetRawMessageFromMAKE(11,IsMAKERunning);
  ShowMessage(IsMAKERunning);
end;
```

Note: Reading a message can be a little bit slow, so only read the messages that you actually care to monitor. If you need more performance, contact scripting@omax.com for additional ideas on higher performance monitoring.

To learn more about Windows Messages, please consult the following sample files.

> GetXYZFromMAKE_Simple.omaxscript
>
> GetXYZFromMAKE_Advanced.omaxscript
>
> GetMessagesFromMAKE.omaxscript
>
> MonitorAllMessages.omaxscript

WINDOWS MESSAGE ID NUMBERS

The following ID numbers can be passed in as IDNumber into OMAX_GetRawMessageFromMAKE to return the documented results below.

0	Protocol version (What version of this message structure is being sent. This will be “2” or greater.)
1	Time stamp (Similar to OMAX_GetTickCount, this is the TickCount at which the last status information was received, which can be used to determine how fresh the data is.)
2	Byte that counts by 1 every time status is read from the OMAX USB device by the device driver. (Use to know that status is working – typically this just counts from 0 to 255 over and over. Think of this as a “heart beat” to indicate that the control system is alive.)
3	Linecounter indicating the Frame Number Location in tool path that is presently being fed by the OMAX USB device. See also #17 to get the total number of frames in the path.
4	Status Flags, expressed as a byte (Stopped, decelerating, baby sit status, etc.) Bit 4 = Write Enabled (USB card can accept data) Bit 5 = Fault flag (something is wrong) Bit 6 = Decelerating (A stop is commanded, we are decelerating) Bit 7 = Stopped Flag
5	Error Flags, expressed as a byte, to indicate why we are stopped. Bit 0 = Stopped due to a Fault Bit 1 = End of the tool path is in the USB card buffer (nearing end of path) Bit 2 = Stopped from buffer low Bit 3 = Mid stop on abrasive (a stop was commanded while an abrasive event was in process.) Bit 4 = Debug flag (used in firmware debugging)
6	Output bits (Abrasive, jet, etc.) See OMAX Tester for definitions. Bit 0 = Pump (1 = 0n / 0 = off) Bit 1 = Low Pressure (1 = enable low pressure mode) Bit 2 = Jet (nozzle) (1 = on / 0 = off) Bit 3 = Abrasive (1 = on / 0 = off) Bit 4 = Hard stop homing (1 = enabled / 0 = disabled) Bit 5 = Terrain Follower for Z axis #1 (1 = enabled / 0 = disabled) Bit 6 = Drill (1 = on / 0 = off) Bit 7 = Drill Down (1 = send it down, 0 = send it up) Bit 8 = Accessory bit (reserved) Bit 9 = Terrain Follower for Z axis # 2 (1 = enabled / 0 = disabled) Bits 10 to 15 = (reserved)
7	Output bit mask value – Low AND byte (Used to report status of masks that may override the above output bits.)
8	Output bit mask value – Low OR byte
9	Output bit mask value – High AND Byte
10	Output bit mask value – High ORD byte
11	Is MAKE running an actual tool path that could cut? (1 = True, 0 = False)
12	MAKE's Absolute (relative to absolute home) X-coordinate position in motor steps (See also OMAX_GetXYZFromMAKE to get this in inches)

13	MAKE's Absolute (relative to absolute home) Y-coordinate position in motor steps (See also OMAX_GetXYZFromMAKE to get this in inches)
14	MAKE's Absolute Z1-coordinate position in motor steps (See also OMAX_GetXYZFromMAKE to get this in inches)
15	MAKE's Absolute Z2-coordinate position in motor steps (See also OMAX_GetXYZFromMAKE to get this in inches)
16	MAKE's Rotary Axis position in integer units of 1/10,000th of a degree
17	Total Length of the bitstream loaded in MAKE in units of “frames”
18	Total Elapsed Time of the presently active path, in milliseconds
19	Total Elapsed Time of the presently active path that it has been cutting, in milliseconds
20	Reserved for OMAX internal use only.
21	Percent of path completed, as expressed in distance (not time)
22	Is MAKE's VCR form (the dialog used when cutting paths) visible? (1 = True, 0 = False)
23	Indicates if pressure units are expressed as “1=Pressure at pump”, or “0=Pressure at nozzle”
24 – 34	Unavailable for scripting
35	Quality of current item being cut in the tool path if MAKE is cutting ('0' is reported if MAKE is not cutting)
36	Unavailable for scripting
	Information from Counters and Timers (similar to CountersAndTimers.INI, but live)
37	Pump time since the start of the cutting job in milliseconds
38	Time at low pressure since the start of the cutting job in milliseconds
39	Time with Jet on since the start of the cutting job in milliseconds
40	Time with Abrasive On since the start of the cutting job in milliseconds
41	Pump cycles since the start of the cutting job
42	Low Pressure cycles since the start of the cutting job
43	Jet on cycles since the start of the cutting job
44	Abrasive on cycles since the start of the cutting job
45	Fault information (Details on any fault condition – email scripting@omax.com for additional documentation.)
46	Additional fault information (Details on any fault condition – email scripting@omax.com for additional documentation.)
47	Last error code from the tool path interpolator (Used for automated testing)
48	Interpolator error level (Used for automated testing)
49	Percent override if a feedrate override is used (Requires USB firmware version 62 and higher)
50	Cycles remaining (if VCR control is showing). Shows how many cycles are remaining in the VCR dialog.
51	High byte flagging which axes are attached (email scripting@omax.com for details.)
52	Low byte flagging which axes are attached (email scripting@omax.com for details.)
53 - 123	<RESERVED FOR FUTURE USE>
124	Total Pump time for the machine, in milliseconds
125	Total Time spent by the machine at Low Pressure, in milliseconds
126	Total Time for the machine with the Jet On, in milliseconds
127	Total Time for the machie, with the Abrasive On in milliseconds
129 - 257	Each element in this section is used to store a character that list the complete file path of the file that is loaded into MAKE. 257 contains a null character.

258 +	<RESERVED FOR FUTURE USE>

Notes:

- Many of the items above can typically be ignored, as they are for specialty purposes beyond the scope of normal scripting use. Read only the items that are necessary for the task at hand.
- These constanst are defined in a unit of code that you can include in your project, like so:

  ```
  uses
    '%OMAX_Library%OMAX_Messaging.omaxscript';
  ```

  This unit also contains functions that may also be of use.

- Example scripts on messaging can be found in the …\Demos_and_Samples\Concepts_for_Learning\Messages_and_Monitoring folder.

OMAX. (OMAX "DOT") CAD OBJECT REFERENCING

In addition to the OMAX_ commands, it is also possible to make some references directly to the OMAX CAD Component Object. Referencing the CAD object directly is done by typing "OMAX" followed by a period ".", then the CAD property in question.

Properties can be read, as in:

```
OldDotStatus:=OMAX.Dots;
```

...and many properties can be written / set as in:

```
OMAX.Dots:=OldDotStatus;
```

Note: Reading and writing of properties can cause certain events to occur. For example, writing the "Dots" property will cause the screen to be redrawn in the new state.

Note: When you set a property, it will typically remain set even after the script has finished running, so use with caution.

Some OMAX Object properties of interest or use are listed below:

BitmapFileName: String
This sets the bitmap image to use as the background (as is used by the OMAX Bitmap Tracing features.) Use with:

ShowBitmap: Boolean
Determines whether or not to show a bitmap specified with BitmapFileName.

ColorScheme: Integer
An integer value that determines which color scheme to use. The default of black is zero.

DivideOnSnap: Boolean
Determines whether or not entities should be divided when a snap occurs on them

Dots: Boolean
Enables or disables showing white endpoint dots.

DrawColor:TColor (Integer)
This is the color used for rubber-banding

DrawPolygonConstructionLine:Boolean
Determines whether or not to draw an extra line from the geometric center of a polygon to it's corner when generating polygons.

DrawThickness: Integer
Sets the thickness in pixels at which all lines are drawn at when *rubber-banding*.

EntityDrawThickness
Sets the thickness in pixels by which all other entities are drawn (so fattens up the entire drawing.)

LeadIOVDegrees: Double
LeadIOVErrorColor: TColor (Integer)
LeadIOVForceSize: Double
LeadIOVMaxLength: Double
LeadIOVMaxLengthOUT: Double
LeadIOVMinLength:Double
LeadIOVMinLengthOUT:Double
LeadIOVOffsetRight:Boolean
The above all determine the behavior of the Lead generation commands.

LocalZeroSize: Integer
Size in pixels of the crosshair used to show the local Zero point of the drawing.

ZeroPointColor:TColor (Integer)
The color of the local zero point crosshair.

Ortho: Boolean
Enables or disables Ortho mode

OrthoDegrees:Double
Number of degrees to lock angles to when Ortho is enabled.

ScaleMax:Double
ScaleMin:Double
Sets the maximum and minimum scale factor one is allowed to zoom to.

TrimFillet:Boolean
If True, then extra entities are erased when the fillet command is used, if False, they are left there.

Width: Integer
Height: Integer
Top: Integer
Left: Integer
The location and size of the CAD box in pixels

Advanced Note: Since the OMAX CAD Object inherits from a standard Delphi TPaintBox object, other OMAX object properties can be researched by searching for the Delphi TPaintBox object on the Internet. This might have some obscure application, but it is generally recommended to stay away from this unless you are an advanced programmer.

TCADBOXSNAP OMAX OBJECT.

The possibilities of scripting run very deep. The advanced script writer may wish to explore such depths with the TCadBoxSnap OMAX Object, which is a complete CAD box in a single component. In fact, it is the same CAD box that is used in many places throughout the OMAX software for display and CAD purposes.

Using a TCadBoxSnap object, it is possible to include additional CAD objects within a script. These, in turn, can execute their own scripts, allowing for CAD within CAD, and Scripts within Scripts.

Huh?

Let’s say you are creating a dialog where you want to give the user a preview of what the results of the edit boxes in the dialog will do after they press “OK”. In this case, you need a second CAD box to display things in, that you can stick in your user interface, and you will need to be able to have control over where it is, how it is displayed, and what it contains. A powerful way to do this is to create a second instance of the CAD system on your form, and use that to run another script that generates the preview.

(This is how the “Tab / Bridge” command in LAYOUT generates the preview in its user interface.)

For an example of this in action, explore the CAD_Inside_Of_CAD.omaxscript Sample script.

As demonstrated in CAD_Inside_Of_CAD.omaxscript, the TCADBOXSNAP object has two properties which are used to send commands to the object: ScriptToRun, and ScriptCommand.

ScriptToRun

This property allows you to trigger a script which will run in the CAD box by assigning a string to this property. The string can contain an absolute path to the script, or a relative one.

```
MyCadBox.ScriptToRun:='MyScript.omaxscript';
```

or

```
MyCadBox.ScriptToRun:='C:\long\path\to\MyScript.omaxscript';
```

ScriptCommand

This property allows you to trigger individual commands in a CAD Box by assigning a string to this property. You may pass in a single command, or a long string or commands separated by semicolons.

```
MyCadBox.ScriptCommand:='OMAX_E_SetX1(2, 20); OMAX_E_SetX2(2, 40)';
```

Note: The syntax must be in PascalScript. Support for other languages in ScriptCommand is not available. ScriptCommand is intended for simple operations, if you need something more complex, use ScriptToRun, or call functions from another “used / included” unit.

Be careful to check the syntax of your string, since the line:character error message won’t be of much help (it will always say line 3). You may not define variables within the command string as there isn’t any place to declare variables. For adding long, complex scripts to CAD boxes, write your script on another file and use ScriptToRun instead.

To use quotes inside of your command string, you must use ASCII symbol #39.

```
MyString:=’hello’;
```

```
MyCadBox.ScriptCommand:='ShowMessage('+#39+MyString+#39+');';
```

Only use the ScriptToRun and ScriptCommand properties of CAD Box objects which you have created, and not the OMAX CAD Box from which you are running the command

```
OMAX.ScriptToRun:=’MyScript.omaxscript’; // will cause an error
```

SCRIPT COMMAND REFERENCE

There are example scripts demonstrating most of the supported commands. These will be in the folders indicated for each command under the default folders shown here:

C:\Program Files (x86)\OMAX Corporation\OMAX_LAYOUT_and_MAKE\
Demos_and_Samples\Concepts_for_Learning

APPENDIX 1 - FORMS CONTROLS

The following classes can be accessed inside a script:

TControl
property TControl.Parent
procedure TControl.Hide
procedure TControl.Show
procedure TControl.SetBounds(ALeft, ATop, AWidth, AHeight: Integer)
event TControl.OnCanResize
event TControl.OnClick
event TControl.OnDblClick
event TControl.OnMouseDown
event TControl.OnMouseMove
event TControl.OnMouseUp
event TControl.OnResize

TWinControl
procedure TWinControl.SetFocus
event TWinControl.OnEnter
event TWinControl.OnExit
event TWinControl.OnKeyDown
event TWinControl.OnKeyPress
event TWinControl.OnKeyUp

TCustomControl
TGraphicControl
TGroupBox
TLabel
TEdit ***
TMemo

TCustomComboBox
property TCustomComboBox.DroppedDown
property TCustomComboBox.ItemIndex

TComboBox
TButton
TCheckBox
TRadioButton

TCustomListBox
property TCustomListBox.ItemIndex
property TCustomListBox.SelCount
property TCustomListBox.Selected

TListBox
TControlScrollBar
TScrollingWinControl
TScrollBox

TCustomForm
procedure TCustomForm.Close
procedure TCustomForm.Hide
procedure TCustomForm.Show
function TCustomForm.ShowModal: Integer
event TCustomForm.OnActivate
event TCustomForm.OnClose
event TCustomForm.OnCloseQuery
event TCustomForm.OnCreate
event TCustomForm.OnDestroy
event TCustomForm.OnDeactivate
event TCustomForm.OnHide
event TCustomForm.OnPaint
event TCustomForm.OnShow
property TCustomForm.ModalResult

TForm

type TModalResult
type TCursor
type TShiftState
type TAlignment
type TAlign
type TMouseButton
type TAnchors
type TBevelCut
type TTextLAYOUT
type TEditCharCase
type TScrollStyle
type TComboBoxStyle
type TCheckBoxState
type TListBoxStyle
type TFormBorderStyle
type TWindowState
type TFormStyle
type TBorderIcons
type TPosition
type TCloseAction

***
```
Frequently you will be checking a
field that is 1 of a group, but you
need to know which field is being
updated.
One way is to use an OnKeyUp action,
set up like this:
```

```
(MyEditField[1] and [2] are TEdit
fields)
…
MyEditField[1].Tag:=1;
MyEditField[1].OnKeyUp    :=
@CheckEntry;
…
MyEditField[2].Tag:=2;
MyEditField[2].OnKeyUp    :=
@CheckEntry;
…

procedure CheckEntry(SenderObj:
TObject; var Key: Word; Shift:
TShiftState);
var
  TempField : TEdit;

begin
  // Type cast SenderObj as a TEdit
  // This will work as long as ONLY
1 type of field is calling this
procedure
  TempField := MyEditField
[TEdit(SenderObj).Tag];

  // Show text from the field that
called this procedure.
  ShowMessage(TempField.Text);

  …
end;
```

APPENDIX 2 - FORMS EXTENDED CONTROLS

The following classes can be accessed inside a script:

TShape
TPaintBox
event TPaintBox.OnPaint
TImage
TBevel
TTimer
event TTimer.OnTimer
TPanel
TSplitter
TBitBtn
TSpeedButton
TCheckListBox
property TCheckListBox.Checked
TTabControl
TTabSheet
TPageControl
procedure TPageControl.SelectNextPage(GoForward: Boolean)
property TPageControl.PageCount
property TPageControl.Pages
TStatusPanel
TStatusPanels
function TStatusPanels.Add: TStatusPanel
property TStatusPanels.Items
TStatusBar
TTreeNode
procedure TTreeNode.Delete
function TTreeNode.EditText: Boolean
property TTreeNode.Count
property TTreeNode.Data
property TTreeNode.ImageIndex
property TTreeNode.SelectedIndex
property TTreeNode.StateIndex
property TTreeNode.Text
TTreeNodes
function TTreeNodes.Add(Node: TTreeNode; const S: string): TTreeNode
function TTreeNodes.AddChild(Node: TTreeNode; const S: string): TTreeNode
procedure TTreeNodes.BeginUpdate
procedure TTreeNodes.Clear
procedure TTreeNodes.Delete(Node: TTreeNode)
procedure TTreeNodes.EndUpdate
property TTreeNodes.Count
property TTreeNodes.Item
TTreeView
procedure TTreeView.FullCollapse
procedure TTreeView.FullExpand
property TTreeView.Selected
property TTreeView.TopItem
TTrackBar
TProgressBar
TListColumn
TListColumns
function TListColumns.Add: TListColumn
property TListColumns.Items
TListItem
procedure TListItem.Delete
function TListItem.EditCaption: Boolean
property TListItem.Caption
property TListItem.Checked
property TListItem.Data
property TListItem.ImageIndex
property TListItem.Selected
property TListItem.StateIndex
property TListItem.SubItems
TListItems
function TListItems.Add: TListItem
procedure TListItems.BeginUpdate
procedure TListItems.Clear
procedure TListItems.Delete(Index: Integer)
procedure TListItems.EndUpdate
property TListItems.Count
property TListItems.Item
TIconOptions
TListView
TToolButton
TToolBar
TMonthCalColors
TDateTimePicker
TMonthCalendar
type TShapeType
type TBevelStyle
type TBevelShape
type TResizeStyle
type TButtonLAYOUT
type TButtonState
type TButtonStyle
type TBitBtnKind
type TNumGlyphs
type TTabPosition
type TTabStyle
type TStatusPanelStyle
type TStatusPanelBevel
type TSortType
type TTrackBarOrientation
type TTickMark
type TTickStyle
type TProgressBarOrientation
type TIconArrangement

type TListArrangement
type TViewStyle
type TToolButtonStyle
type TDateTimeKind
type TDTDateMode
type TDTDateFormat
type TDTCalAlignment
type TCalDayOfWeek

APPENDIX 3 - FORMS DB CONTROLS

TDBEdit
TDBText
TDBCheckBox
property TDBCheckBox.Checked
TDBComboBox
property TDBComboBox.Text
TDBListBox
TDBRadioGroup
property TDBRadioGroup.ItemIndex
property TDBRadioGroup.Value
TDBMemo
TDBImage
TDBNavigator
TDBLookupControl
property TDBLookupControl.KeyValue
TDBLookupListBox
property TDBLookupListBox.SelectedItem
TDBLookupComboBox
property TDBLookupComboBox.Text
TColumnTitle
TColumn
TDBGridColumns
function TDBGridColumns.Add: TColumn
property TDBGridColumns.Items
TDBGrid
type TButtonSet
type TColumnButtonStyle
type TDBGridOptions