Dazzle Technologies Corp.
403 Glenn Dr., Suite 11
Sterling, VA 20164 USA
Phone: (703) 430-8812 Fax: (571) 434-7899

Excerpt from *Extending Authorware*, by Joseph Ganci and Christopher Swenson

Authorware, Heal Thyself – Use Authorware Scripts to Fix and Change Your Files
(excerpt)

Authorware has many functions that you can use to create scripts that will dive through your code, hunting for specific targets and change them. These scripts are often called 'bots, short for robots. They go through your code, hunt for specific targets you specify, and return information to you, change your code, or both.

1. These code hunters can save you an enormous amount of time. Correctly written, the investment in their creation pays back in dividends because it allows you to avoid the monotonous task of having to update or change multiple instances of code one by one. This is one way that experts distinguish themselves from novices. While the novice is laboring away repeating the same task over and over, the expert writes a script once, and lets it go to town.
2. Novices should not feel as if experts are hiding some big secret from them. The reason novices aren't simply told how to create these 'bots is that they are difficult to write before a good deal of understanding of Authorware scripting is achieved. Are you ready? Ready or not, here we go!
3. So what is the mystery? The whole secret behind creating a 'bot is to take advantage of Authorware's functions and variables that tap into the properties of each icon and to understand the relationships between icons. It has always been possible to access *some* of the properties of Authorware icons through a script. For example, even in version 1 of Authorware, you could use:

   DisplayIcon(IconID@"my turtle")
4. The above allows you to show the contents (one of the properties) of the "my turtle" icon from any script in the file.
5. With each new version of Authorware, more and more properties have become accessible. Authorware 6 brought us a bonanza of new open properties, which allows us to write 'bots like never before. Many of the ones used in creating Knowledge Objects and Commands (discussed later) can be used to create simple script 'bots as well. As your 'bots are meant to run as programmer aids, that means they will be set up to run only in source mode, which means you are not restricted by the rule that some of the functions can't be run in packaged mode. In other words, you have full access to everything that is accessible.
6. Some but not all of the functions that can be useful in the creation of 'bots include the following.

Working with icons:	**ClearIcons** **CopyIcons** **CutIcons** **GetLibraryInfo** **GetPasteHand** **GetSelectedIcons** **GroupIcons** **ImportMedia** **InsertIcon** **PasteIcons** **PasteModel** **SelectIcon** **SetIconTitle** **UngroupIcons**
For manipulating Calc scripts:	**GetCalc** **GetFunctionList** **GetInitialValue** **GetVariable** **GetVariableList** **SetCalc**
For checking icon contents:	**GetIconContents**
For checking with whole file:	**GetFileProperty** **IsCourseChanged** **IsLibraryChanged** **OpenFile**

	OpenLibrary **PackageFile** **PackageLibrary** **SaveFile** **SaveLibrary**

7. Often, a client will come back to you after you have finished an 80-lesson course, where each lesson is in its own separate file. The client may then request a change that requires you to enter each lesson and make changes, often to multiple places in each file. This invariably leads to the following:

   - The programmer spends a good deal of time doing very monotonous (and expensive) work.
   - Being human, the programmer will miss one or more places where the change was to take place.
   - A total Quality Assurance (QA) cycle must take place on each lesson to ensure that the programmer didn't miss anything or didn't cause more problems to occur through a late-night slip of the mouse.

8. While you can never totally avoid QA cycles, correctly writing a 'bot to make the changes for you, when possible, will lead to smoother QA reports since a correctly written 'bot will:

   - Perform the work very quickly.
   - Not miss any place where the change was to take place (if correctly written).
   - Not complain about working late nor demand a raise.

9. Let's create a simple 'bot. We have just received a set of Authorware files from a new client. The client tells us, "These files have been around for a while and the company that originally created them has now been bought out by Bill Gates and subsequently was closed. Every programmer has mysteriously vanished without a trace. We need you to make changes to these files."

10. The client has asked that to start, every Calculation icon in each file should include a running set of comments showing the history of changes made to the script it contains. This is a good idea that expert programmers often use. The history might show, for example:

```
-- 4/22/2002 Pedro Gonzalez
-- I changed all of the occurrences of "mean old boss" to "kindly
 --despot"
-- 4/23/2002 Susan Orbitz
-- "kindly despot" was rejected in favor of "respected tyrant". Changes have been made.
```

11. This kind of history can be very helpful in tracking changes (and knowing who to blame ☺). The client has requested that we start this commentary by adding the following lines to the end of *every* script in *every* Calc icon:

```
-- [Today's Date] Start of History
-- Please place any changes you make to this script below
-- Place the date, then your name on one line
-- Follow it with your comments on subsequent lines
```

12. We *could* hunt down every Calculation icon in each file and paste the above four lines at the end of each script, but we're a bit nervous about missing some of the Calculations, especially those that are ornaments to other icons, because the client has threatened us with nonpayment if we miss even one Calculation icon (pretty mean, huh?).

13. To start, we know we're going to need to dive through the file and find every Calculation icon. To do this, we will use a *dive* script. Macromedia has provided one that we can use.

14. Second, we know we're going to need to:

   - Touch every Calculation icon,
   - Grab its contents,
   - Add our new lines to the end of the contents, and
   - Place the script back into the Calc icon.

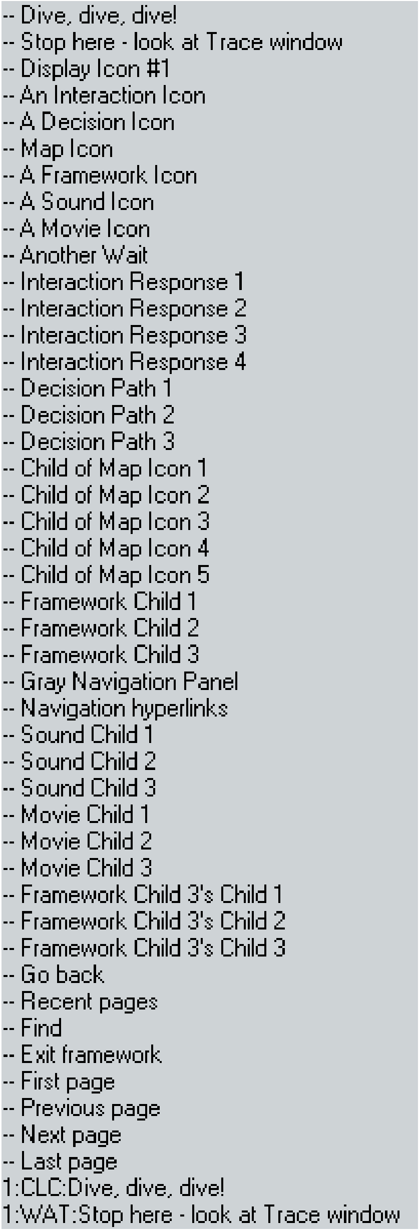
15. That should do it! Let's start with a slightly modified version of the standard Macromedia dive routine.

```
Dive[#BranchList] := [RootIcon ^ ", 0"]

repeat while ListCount(Dive[#BranchList]) > 0
    Dive[#ParentIcon] := GetNumber(1, Dive[#BranchList][1])
    repeat with Dive[#ChildNum] := 1 to IconNumChildren(Dive[#ParentIcon], GetNumber(2, Dive[#BranchList][1]))
          Dive[#ChildIcon] := ChildNumToID(Dive[#ParentIcon], Dive[#ChildNum], GetNumber(2, Dive[#BranchList][1]))
       if IconType(Dive[#ChildIcon]) = 4 |=5 |=6 |=9 |=10 then
           AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 0")
       else if IconType(Dive[#ChildIcon]) = 12 then
           AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 0")
           AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 1")
           AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 2")
        end if

-------------------------------- Do custom stuff here -------------------------

    if GetCalc(Dive[#ChildIcon]) <> "" then
          AddLinear(Calcs, Dive[#ChildIcon])
    end  if
---------------------------------- end of doing stuff ----------------------------
    end repeat
    DeleteAtIndex(Dive[#BranchList], 1)
end repeat
```

16. The above script starts at the root of the file and finds every icon. It does not automatically create a list of each icon. Instead, it presents each icon in turn at the start of the comment line "Do custom stuff here" at which point you can deal with the icon any way you wish. The icon is presented by its ID number, which is stored in the property list index **Dive[#ChildIcon]**.

17. Note how the dive occurs.
    - The *BranchList* property of the *Dive* list keeps a running list of any icons that are encountered.
    - Once an icon has been checked, its id is deleted from the list. Think of it as a First In, First Out (FIFO) line. As people line up outside a movie theater, the person at the front of the line pays for his ticket, and then enters the theater, therefore disappearing from the line. The last person in the line is the last to disappear from the line. As each person reaches the front of the line, the question arises as to whether that person has children. If children exist, they get added to the end of the line. If those children have children in turn (in essence the grandchildren of the first person), they also get added at the end of the line when their parents reach the front of the line.
    - Similarly, the dive code uses a FIFO approach. Whenever a Map, Framework, Interaction, Decision, Digital Movie or Sound icon is found, it gets added to the list so that its children can be later checked as well. In the case of all parent icons except Frameworks, the icon is added with a ",0", which will indicate that we should check its children. In the case of Framework icons, however, there are three sets of potential children: those attached to the Framework, those in the Framework's entry pane, and those in its exit pane, so we add a Framework's icon id three times, in each case adding ",0", "1", or ",2" to the end of the icon id.
    - Note that if the icon found is not a parent, the inner repeat loop doesn't occur, as there are no children.
    - The outer repeat loop will keep going *while* the *BranchList* list is not empty. Every time we have finished dealing with an icon, notice at the bottom of the script that it gets deleted from the list (our friend at the start of the line has entered the theater). When there are no more icons left in *BranchList*, then we have finished our dive.
    - The inner repeat loops once for each child found in a parent icon so you can choose to deal with the specific child.

18. An easy way to see how the code touches each icon is to use *Trace(IconTitle(Dive[#ChildIcon]))* in the custom portion of our script. This is an example from a different file.



```
-- Dive, dive, dive!
-- Stop here - look at Trace window
-- Display Icon #1
-- An Interaction Icon
-- A Decision Icon
-- Map Icon
-- A Framework Icon
-- A Sound Icon
-- A Movie Icon
-- Another Wait
-- Interaction Response 1
-- Interaction Response 2
-- Interaction Response 3
-- Interaction Response 4
-- Decision Path 1
-- Decision Path 2
-- Decision Path 3
-- Child of Map Icon 1
-- Child of Map Icon 2
-- Child of Map Icon 3
-- Child of Map Icon 4
-- Child of Map Icon 5
-- Framework Child 1
-- Framework Child 2
-- Framework Child 3
-- Gray Navigation Panel
-- Navigation hyperlinks
-- Sound Child 1
-- Sound Child 2
-- Sound Child 3
-- Movie Child 1
-- Movie Child 2
-- Movie Child 3
-- Framework Child 3's Child 1
-- Framework Child 3's Child 2
-- Framework Child 3's Child 3
-- Go back
-- Recent pages
-- Find
-- Exit framework
-- First page
-- Previous page
-- Next page
-- Last page
1:CLC:Dive, dive, dive!
1:WAT:Stop here - look at Trace window
```

19. We did this by placing the following **one** line in the custom section.

```
Trace(IoncTItle(Dive[#ChildIcon]))
```

20. In our custom code, we're not checking if the icon is a Calculation. Instead we're asking if the contents of the Calculation are not empty. If it isn't, then we will add the icon to our list to modify its contents at the bottom of our script. Note that this will work whether the icon is a Calculation or whether a Calculation ornament is attached to any other icon. Woo-hoo!

21. Once we have finished with our dive, we will have a list called *Calcs* that will contain the icon ID of every icon that is either a Calculation icon or has a Calculation ornament attached to it. We can now apply the following to add our comments to the end of each Calculation script once we run our file.

```
-- change each Calc icon
repeat with i := 1 to ListCount(Calcs)
   SetCalc(Calcs[i], GetCalc(Calcs[i]) ^ "– " ^ Date ^ " Start of History\r" ^ "-- Please place any changes you make to this script below\r"¬
       ^  "-- Place the date, then your name on one line\r-- Follow it with your comments on subsequent lines\r\r")
end repeat
```

22. The above script runs through the Calc list. It uses the *SetCalc* function, which changes the contents of a Calculation icon or ornament. It takes two arguments. The first is the ID number of the Calculation icon to change; the second is the data you wish to

place in the icon. The data in this case we obtain by first using the *GetCalc* function to grab the current contents of the Calculation. That data gets concatenated to the four data lines we wish to add, and the whole thing gets used as the second argument to *SetCalc*.

23. Note that every time you run this script, it will add a new set of comments to the end of every Calculation script. You can further enhance this code by making sure the comments don't already exist in a Calculation before adding them. You can use the *GetCalc* and *Find* functions to do this. This would be a good exercise for you to try.

The file *AddCommentsBot.a6p* contains this script for you to peruse. Add the first Calc to any file. Have fun!

Converting Your Bot to a Knowledge Object

1. Now that you've learned the basics of the dive routine and how to make changes to your icons through a script, it's pretty easy to convert your script to a Knowledge Object, so that it can be dropped into any file and work well.

2. To convert the script to a Knowledge Object is not at all difficult. We just have to keep in mind that we are not going to be looking at Calculation icons to change in the current file, but in a target file. This means that when we drop our Knowledge Object into any Authorware source file, it will work on that source file.

3. To accomplish this, we will use Authorware's *CallTarget* function, which is found in the general category. Using the *CallTarget* function allows us to use any of the other functions that are found in the *Target* category.

4. The only real trick is, what parts of our script should be pointing to the target file? The answer is any that the target file needs to address.

5. Here is the script, now modified to work our Knowledge Object.

```
Dive[#BranchList] := [CallTarget("GetVariable","RootIcon") ^ ", 0"]

repeat while ListCount(Dive[#BranchList]) > 0
 Dive[#ParentIcon] := GetNumber(1, Dive[#BranchList][1])

 repeat with Dive[#ChildNum] := 1 to CallTarget("IconNumChildren",Dive[#ParentIcon], GetNumber(2, Dive[#BranchList][1]))
  Dive[#ChildIcon] := CallTarget("ChildNumToID",Dive[#ParentIcon], Dive[#ChildNum], GetNumber(2, Dive[#BranchList][1]))
  if CallTarget("IconType",Dive[#ChildIcon]) = 4 |=5 |=6 |=9 |=10 then
   AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 0")
  else if CallTarget("IconType",Dive[#ChildIcon]) = 12 then
   AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 0")
   AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 1")
   AddLinear(Dive[#BranchList], Dive[#ChildIcon] ^", 2")
  end if

-------------------------------- Do custom stuff here ----------------------------

  if CallTarget("GetCalc",Dive[#ChildIcon]) <> "" then AddLinear(Calcs, Dive[#ChildIcon])

---------------------------------- end of doing stuff -----------------------------
 end repeat
 DeleteAtIndex(Dive[#BranchList], 1)
end repeat

-- change e ach Calc icon
repeat with i := 1 to ListCount(Calcs)
 CallTarget("SetCalc",Calcs[i], CallTarget("GetCalc",Calcs[i]) ^ "-- " ^ Date ¬
  ^ " Start of History\r-- Please place any changes you make to this script below\r" ¬
  ^ "-- Place the date, then your name on one line\r– Follow it with your comments on subsequent lines\r\r")
end repeat
```

6. In the example file we are providing with this document, we have provided for a little message to appear on the screen and a Done button for the user to click, after which the file will quit.

7. To make this application work, we need to do the following:

   a. Package without runtime the above Authorware file and place the packaged file in a subfolder of the Knowledge Objects folder (inside the Authorware 6 folder). Have the subfolder called My KOs or something else you'll remember. Call the packaged file Insert Script Comment.
   b. Click the bottom of the flowline so that the insertion hand is sent there.
   c. Go to the Insert dropdown menu, choose Icon, then choose Knowledge Object Icon. This will insert a new KO icon on the flowline. Title it Insert New Comments.
   d. Open the KO icon. You will get a message saying no wizard was specified and asking if you want to do it now. Click OK.
   e. Click the … button next to the Wizard field. Point it to the packaged file you just created.
   f. Click the Run Wizard checkbox.
   g. Click t he OK button.
   h. Make sure the KO icon is selected, then choose the File pulldown menu, then Save in Model. Save it to the folder of the packaged file as Insert Script Comment.a6d.
8. Now you're ready to go. Open the Knowledge Objects window under the Window pulldown menu. Click the Refresh button and pulldown the categories. You should see the My KOs category and in that category you should see the Insert New Commands KO.
9. Drag the KO over to the flowline and let it do its work!

Converting Your Knowledge Object to a Command

1. This is so easy! Just copy the packaged wizard file you created for the KO into the Commands folder in the Authorware 6 folder.
2. To see the new command and use it, you'll have to close Authorware, and then open it again.
3. Choose the new command from the Command dropdown and let it go to town!