How to organize DM scripts as plug-in packages

Tutorial by Bernhard Schaffer and David R. G. Mitchell

Acknowledgement and Disclaimer

This tutorial has been written using Digital Micrograph version GMS 1.5.1. It is not an official documentation and comes without any warranty. This text has also not been approved by Gatan Inc.

Some of the ideas and scripts within this tutorial were originally developed/posted by Vincent Hou (Micron Tech.) and Werner Grogger (FELMI/ZFE).

Introduction

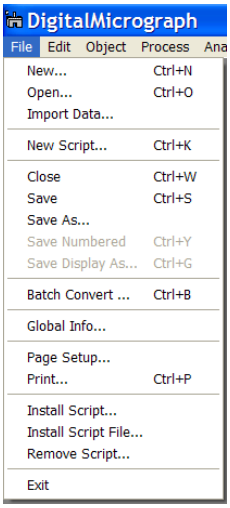
Scripting greatly enhances the capabilities of Digital Micrograph. Whether your write scripts yourself, or simply use those from the Digital Micrograph Scripting Database, sooner or later script management will become an issue for you. For example keeping your menus tidy and their structure consistent, making sure you have the most current versions of particular scripts and sharing script setups between PCs can be problematic.

Basic script installation is covered in the first part of this tutorial. When you install scripts, they are added as menu items, in the order in which they were installed – first at the top of the menu, last at the bottom. If you wish to update a script, you must first delete the old version then install the new. The problem here is that the updated script will be appended at the end of the menu list in which it is installed, thus changing the order in which the menu commands appear. This can be annoying and confusing for users. Ideally, you want to be able to update individual scripts while keeping the overall appearance of your menus constant. You may also wish to include spacer bars in menus to break up large lists of commands and group them logically.

Both of these can be achieved by creating packages of scripts and installing them as plug-ins. A further advantage of script packages (plug-ins) is that it is possible to share the functionality of the package with users without necessarily disclosing the source code. This may be a consideration for commercial developers or those not wishing to disclose proprietary code information. The creation of script package plug-ins is covered in the second part of the tutorial.

Finally, keeping track of script updates can be challenging. Many of us release numerous updates to our scripts in response to bug reports or requests/suggestions from users. With many dozens if not hundreds of scripts installed, knowing which versions are installed and what should be updated can be a major challenge. Here again, package installers come to the rescue. By time coding scripts and storing them in a central server location, maintaining currency becomes very straightforward. This is covered in the final part of this tutorial.

Installing Scripts as Menu Commands / Libraries



Digital Micrograph (*DM*) scripts (*scripts*) are usually launched by opening them in a text window within DM, placing the blinking text cursor somewhere within the window, and pressing the keys CONTROL and RETURN simultaneously. Alternatively, scripts can be "installed" as menu-commands using one of the menu-commands:

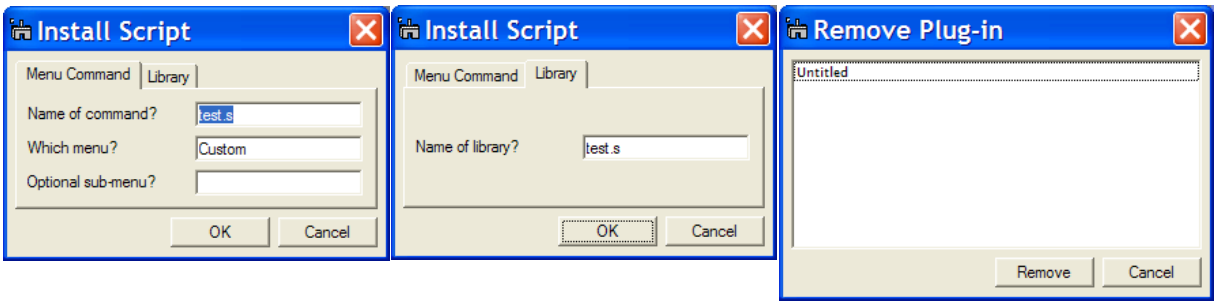
- `File/Install Script…`
- `File/Install Script File…`

The script is then available under a chosen name within a chosen menu/submenu, and can be run by selecting the menu command as you would any other DM menu command. The script code itself is stored within the preferences-file of DM[1] , which is usually located at

C:\Program Files\Gatan\DigitalMicrograph\Prefs\DigitalMicrographCF.8.prf

To remove a script from the menu, call the menu command and identify the script to be removed:

- `File/Remove Script…`



Functions or procedures written in the scripting language can also be installed as libraries, using the same menu commands but selecting the second tab "Library". Libraries are loaded when DM launches. Functions and procedures stored within these libraries are then available as new script commands for all scripts subsequently run on DM. Library commands are in-effect user extensions of the DM scripting language. They eliminate the need to include the code for the functions in the every script which calls them, greatly simplifying code. However, be aware that a script which calls a library function will not be transportable to another installation of DM, unless it too has the library function installed. A package installer which creates both the package of scripts and installs the relevant library files is a good way of avoiding this pitfall.

Note, that it is not possible to have two or more library functions/procedures with the same name and set of parameters installed at the same time. It is thus good programming style to give library functions a unique and easily recognizable name, e.g. with a short prefix common for all routines of one library[2].

[1] Tip: There are two preference files in the above folder – the .8 preference file which stores all the user-added scripts and the .7 preference file which stores the settings of DM. If you want to copy a setup of installed scripts from a source PC to another (target) PC, ensure that DM is not running on the target PC. Then remove the .8 preference file from the corresponding Prefs folder on the target PC (don't delete it, keep it somewhere safe in case you wish to go back to it). Then copy the new .8 preference file to the Prefs folder on the target PC. When you launch DM on the target PC, it will now have the same menu configuration as the source PC. Another useful tip is to make regular backup copies of both the .7 and .8 preference files – especially if the PC is connected to a TEM. If you suffer a system meltdown, reinstalling the .7 and .8 preference files can save a lot of work reconfiguring DM and reinstalling scripts respectively.

[2] For example if a library file called 'ImgProc_ImageProcessingAddinsLib" contained a series of image processing functions, the functions might be called 'ImgProc_UnsharpMask', 'ImgProc_HoughTransform' etc. Use of lengthy rather complex function names ensures that there is little possibility of using the name of an existing function in DM - which would cause an error at DM launch.

It is also not possible to install two scripts with the same Name/Menu/SubMenu combination. If the source code of a library has errors or uses function names already used before, DM will give an error message at launch. However, the library will nevertheless be installed and stored in the preferences file (.8 preference).

Library files are removed in exactly the same way as installed scripts, using the menu-command shown below.

- `File/Remove Script…`

Library files appear in the list of files to be deleted along with all the other scripts. It can be a little confusing recognising a library file. Script files usually have file names which reflect their corresponding menu command. However, the library file does not add any menu commands and so recalling its name and finding it in a long list of scripts can be difficult. Adding 'Lib' to the end of the file name makes library files readily recognisable. In the example above a fictional library of image processing functions was called 'ImgProc_ImageProcessingAddinsLib'.

Installing Scripts as Menu Commands in a Plug-in – general info

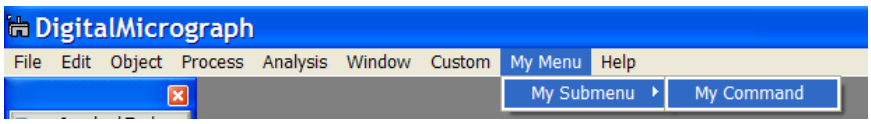
Instead of adding/removing scripts manually from the menu and having them stored in the preferences-file, it is possible to package several scripts together in a plug-in file. These files have the extension *.gtk, *.gt1, *.gt2 or *.gt3 and are stored in the Plug-Ins folder of DM located at:

C:\Program Files\Gatan\DigitalMicrograph\PlugIns\

All files within this folder (but not within subfolder of this folder) are automatically loaded when DM is launched. Scripts stored in packages (plug-ins) then appear as menu commands or are available as library routines[3].

Plug-in files can be created by scripting. The following simple script gives an example of the command syntax for plug-in creation:

```
AddScriptFileToPackage( "C:\\myscripts\\test.s" , "myplugins" , 0 , "MyCommand" , "MyMenu" , "MySubMenu" , 0)
```



When this script is executed, it looks at the given path[4] for the given script file:

C:\myscripts\test.s

If this script is found, it will install it as a menu command with the given name (MyCommand), in the given Menu (MyMenu) and the optional submenu (MySubMenu). However, the script itself is not stored in the preferences files. Instead a new plug-in file called (myplugins) is created in the plug-in-folder:

C:\Program Files\Gatan\DigitalMicrograph\PlugIns\myplugins.gt1

The menu commands can now be installed/removed from any DM installation by adding/removing the myplugins.gt1 file in the corresponding Plug-ins folder.

[3] Tip: You can temporarily disable plug-ins by simply creating a folder within the plug-ins folder. Call it something like 'Disabled Plug-ins'. Move any plug-ins you wish to disable to this folder (DM should not be running when you do this). When you next launch DM it will ignore any plug-ins in . . . /Plug-ins/Disabled Plug-ins'. This technique can be useful to track down a plug-in which is causing an error when DM launches.

[4] Note that the backslash sign (\) used in standard path-names is also used as special character in DM strings. Therefore, for each backslash in the path one has to enter a double-backslash (\\) in the given DM string.

The script command AddScriptFileToPackage( ) has two number parameters (both set to 0 in the example above). The first one defines the “package level” and can be 0, 1, 2 or 3. The value of this parameter, sets the plug-in file extension as *.gt1, *.gt2 ,*.gt3 or *.gtk respectively. This controls the order in which plug-ins are loaded when DM is launched. This is important where one plug-in makes calls on functions in a second plug-in. It is essential that the function-containing plug-in is loaded before the plug-in which make a call to it. In other words the function-containing plug-in gets top priority. For this reason, such a plug-in should have an extension priority greater than the plug-in which calls it. If your plug-in does not make any calls to other plug-ins the loading priority is not important.

The second number parameter can be 0 or 1. This value dictates whether the script will be installed as a menu command or a library, respectively.

Note, that scripts installed as library will be executed automatically when DM is launched. This is useful if you want a script to remind you to fill in a logbook, or set the user and specimen ID etc.

Tip: When you are about to create a plug-in the following will help ensure that your plug-in is working properly. Shut DM down, temporarily relocate all the plug-ins to another location so that DM will launch without them. Then create your plug-in. This does two thing: Firstly by creating your plug-in with no other plug-ins loaded you can ensure that your plug-in is not dependent on another plug-in for its correct operation (unless of course that is your intention). Secondly, a relaunch of DM will clean the application’s memory, which is always a good starting point.

A simple script to install all files from a folder as a plug-in

When creating a plug-in containing several scripts, start by gathering together the scripts into a single folder on the hard disk. The following script will then install all scripts within this folder as menu commands maintaining the first level of sub-folders as sub-menus. The filenames are used as command names.

```
// Define the necessary variables
string base,menu,submenu,item,packageNAME,name
number maxitem,i,j,maxfolder
taggroup tgFILES,tgFOLDERS,tg

// Just some starting text in the results window.
result("\n Automatic Installation of all scripts in a folder as Plugin:\n\n")

// First get the default folder. (In this case, the folder last opened within DM)
base = GetApplicationDirectory(2,0)

// Prompt the user with a dialog to choose a folder, with the default folder as first choice.
// If the user cancels the dialog, the script will stop.
If (!GetDirectoryDialog("Please select the folder containing the scripts",base,base)) exit(0)

// Ask the user for a package name
If (!GetString("Name of package file?","",packageNAME)) exit(0)

// Ask the user for a menu name
If (!GetString("Name of menu to install the scripts in","",menu)) exit(0)

// Get all files/folders in the folder as a tag-list
tgFILES   = GetFilesInDirectory(base,1)
tgFOLDERS = GetFilesInDirectory(base,2)

// Install all files from the main folder as menu commands.
// Count Items in the folder
maxitem = tgFILES.TagGroupCountTags()
i = 0

// Loop through all items
while (i<maxitem)
	{
	// get taggroup of item
	tgFiles.TagGroupGetIndexedTagAsTagGroup(i,tg)

	// get name of file
	tg.TagGroupGetTagAsString("Name",item)

	// Only if filename end with ".s" continue
	If (right(item,2)==".s")
		{
		// use the name without the ending
		name = left(item,len(item)-2)
		result("\n Installing: "+item)

		// install the menu command
		// use the Try-Catch loop to detect problems during install
		try
			{
			AddScriptToPackage(base+item,packageNAME,0,name,menu,"", 0)
			}
		catch
			{
			result(" \t ERROR DURING INSTALL")
			}
		}
	i++
	}

// Now install all files from sub-folder as sub-menu commands.
// Count subfolders in the folder
maxfolder = tgFOLDERS.TagGroupCountTags()


// Loop for all subfolders
for (j=0;j<maxfolder;j++)
```

```
{
// get taggroup of item
tgFolders.TagGroupGetIndexedTagAsTagGroup(j,tg)

// get name of subfolder which is also the name of the submenu
tg.TagGroupGetTagAsString("Name",submenu)

// Get all files in the subfolder as a tag-list
tgFILES   = GetFilesInDirectory(base+submenu,1)

// Count Items in the folder
maxitem = tgFILES.TagGroupCountTags()
i = 0

// Loop through all items as before for the main folder
while (i<maxitem)
	{
	tgFiles.TagGroupGetIndexedTagAsTagGroup(i,tg)
	tg.TagGroupGetTagAsString("Name",item)
	If (right(item,2)==".s")
		{
		name = left(item,len(item)-2)
		result("\n Installing <"+submenu+">: "+item)
		try
			{
			AddScriptToPackage(base+item,packageNAME,0,name,menu,submenu, 0)
			}
		catch
			{
			result(" \t ERROR DURING INSTALL")
			}
		}
	i++
	}
}
```

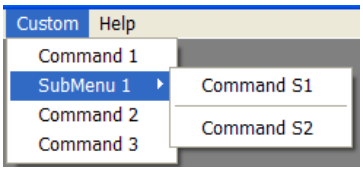
A script template to install various scripts manually – version 1

While it might be convenient to use the previous script for plug-in installation, it is somewhat inflexible. For example, it is often desirable to make the menu command more succinct that the script name, and also to choose the order in which scripts get added to packages (and therefore the order in which commands appear in menus). It is also useful to be able to add separator lines to menus to group scripts. Finally, some fine control over which scripts get added to packages is required when creating varieties of the same package. Examples of this might include packages for novice and expert users, or for offline vs. hardware connected systems. In all these cases a more manual approach is advantageous.

The following script can be used as a template which can be modified or extended as required. Sections which need to be adjusted are marked in blue.

In principle, each individual package requires a “create plug-in script”. The template below is in a simple form. More options and functionality will be added as the tutorial proceeds.

The starting point is to collate together all the scripts to be added to a package into the same folder.

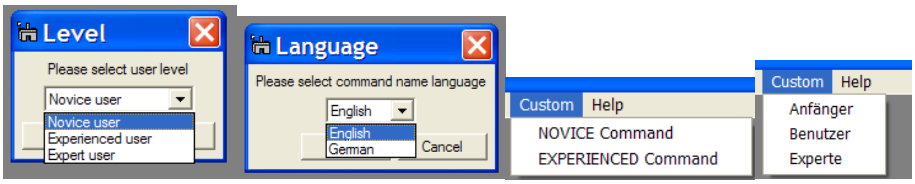



```
//        Install Scripts as plug-in , TEMPLATE
//                        by
//  B.Schaffer (FELMI/ZFE) & D. R. G. Mitchell (ANSTO Materials)
//
//   Acknowledgements: Werner Grogger (FELMI/ZFE)
//                     Vincent Hou (Micron Tech.)
//
//   Template version: 1.0

// As written, this script will install five scripts called Script1.s, Script2.s  - Script5.s into a package
// called ‘My Package’ This creates a series of menu items in the ‘Custom’ menu.

/******* Constants of the package ********/
/*    Please modify this to your needs     */
/*********************************************/

string PackageName      = "My_Package"          // This is the filename of the plug-in
number PackageLevel   = 0                       // This defines the loading priority of
                                                // this plug-in compared to others.
string MenuName           = "Custom"            // This is the default name of the menu
number PackageVersion = 20071121                // This number gives the file version of
                                                // the plug-in. It is convenient to use
                                                // date related numbers of type YYYYMMDD.
string folder                                   // The path to the script containing folder
                                                // including the last backslash (\)
number lineCount      = 0                       // Variable to count number of installed "lines" in the menu.

// Please note that these constants are declared as global variables which
// are therefore also valid within the functions below!


/********* Auxillary Functions  **********/
/*       No modification necessary        */
/*****************************************/
number AddS2P(string ScriptName, string CommandName, string Menu, string SubMenu)
{
  // This function installs a script as a menu command
  result(" Installing <"+ScriptName+">\n\t")
  try
   {
   AddScriptFileToPackage(folder+ScriptName,PackageName,PackageLevel,CommandName,Menu,SubMenu,0)
```

```
  }
 catch
  {
  beep()
  result(" => Error while installing as a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
  return -1
  }
 result(" => Installed as a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
 return 1
}


number AddLib2P(string ScriptName)
 {
 // This function installs a script as a library
 result(" Installing <"+ScriptName+">\n\t")
 try
   {
   AddScriptFileToPackage(folder+ScriptName,PackageName,PackageLevel,ScriptName,"","",1)
   }
 catch
  {
  beep()
  result(" => Error while installing as a library.\n")
  return -1
  }
 result(" => Installed as a library.\n")
 return 1
}


number AddLine2P(string Menu, string SubMenu)
{
 // This function intalls a seperation line in a menu
 // Note, that it is necessary to install each line under a different
 // "position" with regard to menu/submenu/command-name. That is the
 // reason, why the line-count variable is needed.
 LineCount++
 result(" Installing a line\n\t")
 AddScriptToPackage("",PackageName,PackageLevel,"-"+menu+"."+LineCount, menu, submenu, 0)
 result(" => Installed.\n")
 return 1
}

/******* MAIN INSTALLATION SCRIPT ***/
/*    Please modify this to your needs       */
/*********************************************/

result("\n Installing files for plug-in <"+PackageName+">\n")

// First get the base-folder
folder = GetApplicationDirectory(2,0)
If (!GetDirectoryDialog("Please select base folder.",folder,folder)) exit(-1)

// Now install all your scripts. Modify this section manually to your needs.
// The lines shown assume that the scripts are in the base folder.
// They will install a menu called "Custom" (see global parameters) with the
// following structure:
//    CUSTOM
//        + Command 1
//        + SubMenu 1
//                        + Command S1
//                        + ----------
//                        + Command S2
//        + Command 2
//        + Command 3
//
// Additionally, a script "MyLibrary.s" will be installed as a library.

AddLib2P("MyLibrary.s")
AddS2P("script1.s","Command 1",MenuName,"")
AddS2P("script2.s","Command S1",MenuName,"SubMenu 1")
AddLine2P(MenuName,"SubMenu 1")
AddS2P("script3.s","Command S2",MenuName,"SubMenu 1")
AddS2P("script4.s","Command 2",MenuName,"")
AddS2P("script5.s","Command 3",MenuName,"")
```

A script template to install various scripts manually – version 2

This script extends the one described above, by providing user selectable parameters. These enable the functionality of the plug-in to be varied using a single creation script. In this example a plug-in is created for novice, experienced or expert users with the menu items appearing in either German or English. Only the main part of the script has been altered. Unaffected code is marked in gray. The bulk of this code is associated with the creating the dialog.




```
//        Install Scripts as plug-in , TEMPLATE
//                        by
//  B.Schaffer (FELMI/ZFE) & D.R. G. Mitchell (ANSTO Materials)
//
//   Acknowledgements: Werner Grogger (FELMI/ZFE)
//                     Vincent Hou (Micron Tech.)
//
//   Template version: 2.0


/******* Constants of the package ********/
/*    Please modify this to your needs     */
/*********************************************/

string PackageName      = "My_Package"          // This is the filename of the plug-in
number PackageLevel   = 0                       // This defines the loading priority of
                                                // this plug-in compared to others.
string MenuName           = "Custom"            // This is the default name of the menu
number PackageVersion = 20071121                // This number gives the file version of
                                                // the plug-in. It is convenient to use
                                                // date related numbers of type YYYYMMDD.
string folder                                   // The path to the script containing folder
                                                // including the last backslash (\)
number lineCount      = 0                       // Variable to count number of installed "lines" in the menu.

// Please note that these constants are declared as global variables which
// are therefore also valid within the functions below!


/********* Auxillary Functions  **********/
/*       No modification necessary        */
/*********************************************/
number AddS2P(string ScriptName, string CommandName, string Menu, string SubMenu)
{
  // This function installs a script as a menu command
  result(" Installing <"+ScriptName+">\n\t")
  try
   {
   AddScriptFileToPackage(folder+ScriptName,PackageName,PackageLevel,CommandName,Menu,SubMenu,0)
   }
  catch
   {
   beep()
   result(" => Error while installing as a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
   return -1
   }
  result(" => Installed as a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
  return 1
}


number AddLib2P(string ScriptName)
  {
  // This function installs a script as a library
  result(" Installing <"+ScriptName+">\n\t")
```

```
  try
   {
    AddScriptFileToPackage(folder+ScriptName,PackageName,PackageLevel,ScriptName,"","",1)
   }
  catch
   {
   beep()
   result(" => Error while installing as a library.\n")
   return -1
   }
  result(" => Installed as a library.\n")
  return 1
}


number AddLine2P(string Menu, string SubMenu)
{
  // This function intalls a seperation line in a menu
  // Note, that it is necessary to install each line under a different
  // "position" with regard to menu/submenu/command-name. That is the
  // reason, why the line-count variable is needed.
  LineCount++
  result(" Installing a line\n\t")
  AddScriptToPackage("",PackageName,PackageLevel,"-"+menu+"."+LineCount, menu, submenu, 0)
  result(" => Installed.\n")
  return 1
}

/******* MAIN INSTALLATION SCRIPT ***/
/*    Please modify this to your needs       */
/*********************************************/
object      DLG1                        // Object for the dialogues
documentWindow  DLGWindow               // DialogWindow
taggroup DLGtg,DLGitems                 // Items for the dialogues
taggroup DropMenu,DropMenuItems         // DropMenu Items within the dialogues
number userlevel,language               // variables keeping the setting

result("\n Installing files for plug-in <"+PackageName+">\n")

// First get the base-folder
folder = GetApplicationDirectory(2,0)
If (!GetDirectoryDialog("Please select base folder.",folder,folder)) exit(-1)

// Build First Dialog (UserLevel)
DLGtg = DLGCreateDialog("Level",DLGitems)
DropMenu = DLGCreatePopup(DropMenuItems,1)
DropMenuItems.DLGAddPopupItemEntry("Novice user")
DropMenuItems.DLGAddPopupItemEntry("Experienced user")
DropMenuItems.DLGAddPopupItemEntry("Expert user")
DLGitems.DLGAddElement(DLGCreateLabel("Please select user level"))
DLGitems.DLGAddElement(DropMenu)

// Display First Dialog (UserLevel)
DLG1 = Alloc(UIframe).Init(DLGtg)
If (!DLG1.Pose()) exit(-1)

// Get the value (UserLevel)
DropMenu.DLGGetValue(userlevel)

// Build Sceond Dialog (Language)
DLGtg = DLGCreateDialog("Language",DLGitems)
DropMenu = DLGCreatePopup(DropMenuItems,1)
DropMenuItems.DLGAddPopupItemEntry("English")
DropMenuItems.DLGAddPopupItemEntry("German")
DLGitems.DLGAddElement(DLGCreateLabel("Please select command name language"))
DLGitems.DLGAddElement(DropMenu)

// Display Second Dialog (Language)
DLG1 = Alloc(UIframe).Init(DLGtg)
IF (!DLG1.Pose()) exit(-1)

// Get the value (Language)
DropMenu.DLGGetValue(language)

// Now install all your scripts. Modify this section manually to your needs.
AddLib2P("MyLibrary.s")
// The different "languages" lead to different installation trees
```

```
If (language==1)
		{
		If (UserLevel>0) AddS2P("script1.s","NOVICE Command",MenuName,"")
		If (UserLevel>1) AddS2P("script1.s","EXPERIENCED Command",MenuName,"")
		If (UserLevel>2) AddS2P("script1.s","EXPERT Command",MenuName,"")
		}
If (language==2)
		{
		If (UserLevel>0) AddS2P("script1.s","Anfänger",MenuName,"")
		If (UserLevel>1) AddS2P("script1.s","Benutzer",MenuName,"")
		If (UserLevel>2) AddS2P("script1.s","Experte",MenuName,"")
		}
```

A script template to install various scripts manually – version 3

When plug-ins are shared within a broader community, e.g. an institute, it is often problematic to ensure that each PC has the most current version installed. This template is now extended so that each plug-in installs an additional last menu-command called “Plug-in info”, which returns the version number of the actual plug-in. Additionally, the script looks at a hard coded path[5] for newer versions of the package and informs the user. This path will usually be a network path accessible from each installation.

The way this additional functionality works is, that with each version of the plug-in stored at the commonly accessible path, a short script of same name is stored. This script contains only a single line, returning the actual version number as an exit code:

```
Exit(20071120)   // The number is the actual version number of the plug-in
```

Additionally, if the menu command is accessed with the OPTION key held down, then the script also looks for an ASCII-text file at the commonly accessible path and prints it into the results window.

Again the script below is an extension of the template #2. Unchanged text is marked in gray.

```
//        Install Scripts as plug-in , TEMPLATE
//                      by
//  B.Schaffer (FELMI/ZFE) D.R. G. Mitchell (ANSTO Materials)
//
//   Acknowledgements: Werner Grogger (FELMI/ZFE)
//                     Vincent Hou (Micron Tech.)
//
//   Template version: 3.0


/******* Constants of the package ********/
/*    Please modify this to your needs     */
/*********************************************/

string PackageName      = "My_Package"          // This is the filename of the plug-in
number PackageLevel   = 0                       // This defines the loading priority of
                                                // this plug-in compared to others.
string MenuName           = "Custom"            // This is the default name of the menu
number PackageVersion = 20071121                // This number gives the file version of
                                                // the plug-in. It is convenient to use
                                                // date related numbers of type YYYYMMDD.
string folder                                   // The path to the script containing folder
                                                // including the last backslash (\)
number lineCount      = 0                       // Variable to count number of installed "lines" in the menu.

string GlobalPath = "C:\\"                      // This is the path to the network-storage of  all new script-packages.

// Please note that these constants are declared as global variables which
// are therefore also valid within the functions below!


/********* Auxillary Functions  **********/
/*       No modification necessary        */
/*****************************************/
number AddS2P(string ScriptName, string CommandName, string Menu, string SubMenu)
{
  // This function installs a script as a menu command
  result(" Installing <"+ScriptName+">\n\t")
  try
   {
   AddScriptFileToPackage(folder+ScriptName,PackageName,PackageLevel,CommandName,Menu,SubMenu,0)
```

[5] A more flexible solution would be to store the destination path“ in the global tags of each PC. The script then should first look for this path, and only if not found (or the path is non-existent), the hard coded default path would be used. This allows for changing of the network path without recompiling and reinstalling all packages. For simplicity of the code we have not done it that way in this example tutorial.

```
  }
  catch
  {
  beep()
  result(" => Error while installing as a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
  return -1
  }
 result(" => Installed as a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
 return 1
}


number AddLib2P(string ScriptName)
 {
 // This function installs a script as a library
 result(" Installing <"+ScriptName+">\n\t")
 try
  {
  AddScriptFileToPackage(folder+ScriptName,PackageName,PackageLevel,ScriptName,"","",1)
  }
 catch
  {
  beep()
  result(" => Error while installing as a library.\n")
  return -1
  }
 result(" => Installed as a library.\n")
 return 1
}


number AddLine2P(string Menu, string SubMenu)
{
 // This function intalls a seperation line in a menu
 // Note, that it is necessary to install each line under a different
 // "position" with regard to menu/submenu/command-name. That is the
 // reason, why the line-count variable is needed.
 LineCount++
 result(" Installing a line\n\t")
 AddScriptToPackage("",PackageName,PackageLevel,"-"+menu+"."+LineCount, menu, submenu, 0)
 result(" => Installed.\n")
 return 1
}


number AddScriptString2P(string ScriptString, string CommandName, string Menu, string SubMenu, number AsLibrary)
{
 // This function installs a script-string (a script contained in a string variable) as a menu or library command.
 // It installs using the command "AddScriptToPackage" which installs a script (as a string) directly.
 result(" Installing ScriptString ("+CommandName+")\n\t")
 try
  {
  AddScriptTopackage(ScriptString,PackageName,PackageLevel,CommandName,Menu,SubMenu,AsLibrary)
  }
 catch
  {
  beep()
  result(" => Error while installing.\n")
  return -1
  }
 result(" => Installed as ")
 If (!AsLibrary) result("a menu command ("+Menu+"-"+SubMenu+"-"+CommandName+").\n")
 else            result("a library command ("+CommandName+").\n")
 return 1
}

string ReplaceBackSlashByDouble(string IN)
{
 // This function replaces all backslashes in a string by double-backslashes
 // This is needed whenever a string containing real backslashes should be used
 // hard coded in a script, as for example when installing a ScriptLine
 string out=""
 number pos=find(IN,"\\")
 While (pos!=-1)
  {
  out += left(IN,pos)+"\\\\"
  IN = mid(IN,pos+1,len(IN)-pos-1)
```

```
    pos=find(IN,"\\")
   }
  out+=IN
  return out
}

string ScriptGetVersion()
{
  // This function puts together the script for returning the package version.
  // The whole script is contained in one string.
  string out
  out = "number "+PackageName+"_version() return "+Format(PackageVersion,"%8.f")
  return out
  }

string ScriptCompareVersion()
{
  // This function puts together the script for comparing the package version
  // with the net version. The whole script is contained in one string.
  // This script looks kind of strange because all backslashes in paths have
  // to be replaced by double-backslashes if used as hardcoded string in a script
 string out
  out = "void "+PackageName+"_check()\n"
  out += "{\n"
  out += "number instV = "+Format(PackageVersion,"%8.f")+"\n"
  out += "number currV\n"
  out += "result(\"\\n ----------------------------------\")\n"
  out += "result(\"\\n   "+Packagename+"\")\n"
  out += "result(\"\\n ----------------------------------\")\n"
  out += "result(\"\\n  installed version: \"+Format(instV,\"%8.f\")+\"\")\n"
  out += "try \n{\ncurrV = ExecuteScriptFile(\""+ReplaceBackSlashByDouble(GlobalPath+PackageName)+"_v.s"+"\")\n}\n"
  out += "catch \n{\n Throw(\"Error running '"+ReplaceBackSlashByDouble(GlobalPath+PackageName)+"_v.s' \\nPossible reason: file not found.\")\n}\n\n"
  out += "result(\"\\n     newest version: \"+Format(currV,\"%8.f\")+\"\")\n"
  out += "result(\"\\n ----------------------------------\")\n"
  out += "If (currV>instV) OKDialog(\"A newer version of the plug-in is available!\\nnewest version: \"+Format(currV,\"%8.f\"))\n"
  out += "If (currV==instV) OKDialog(\"The installed version of the plug-in is up to date!\\ncurrent version: \"+Format(currV,\"%8.f\"))\n"
  out += "If (currV<instV) OKDialog(\"For some reason the installed version of the plug-in is newer than the one on the network!\\nnetwork version: \"+Format(currV,\"%8.f\"))\n"
  out += "}\n"
  return out
}

string ScriptOutputPackageInfo()
{
  // This function puts together the script for printing the package info stored
  // as a textfile on the network into the results window.
  // The whole script is contained in one string.
  // This script looks kind of strange because all backslashes in paths have
  // to be replaced by double-backslashes if used as hard coded string in a script
  string out
  out = "void "+PackageName+"_info()\n"
  out += "{\n"
  out += "number fileID\n"
  out += "string line\n"
  out += "string path = \""+ReplaceBackSlashByDouble(GlobalPath+PackageName)+"_info.txt\"\n"
  out += "if (!DoesFileExist(path))\n  {\n  OKDialog(\"No information text found.\")\n  exit(0)\n  }\n"
  out += "fileID = OpenFileForReading(path)\n"
  out += "result(\"\\n\\n Current information on package <"+PackageName+">\\n.....\\n\")\n"
  out += "While (ReadFileLine(fileID,line)) result(line)\n"
  out += "CloseFile(fileID)\n"
  out += "result(\"\\n.....\")\n"
  out += "}\n"
  return out
}

string ScriptUpdateButton()
{
  // This function creates the script which is executed with the "Plug-in info" button
  string out
  out  =  "If (OptionDown()) "+PackageName+"_info()\n"
  out +=  "Else "+PackageName+"_check()\n"
  return out
}


/******* MAIN INSTALLATION SCRIPT ***/
/*    Please modify this to your needs       */
/*********************************************/
object      DLG1                          // Object for the dialogues
documentWindow  DLGWindow                 // DialogWindow
taggroup DLGtg,DLGitems                   // Items for the dialogues
```

```
taggroup DropMenu,DropMenuItems     // DropMenu Items within the dialogues
number userlevel,language           // variables keeping the setting

result("\n Installing files for plug-in <"+PackageName+">\n")

// First get the base-folder
folder = GetApplicationDirectory(2,0)
If (!GetDirectoryDialog("Please select base folder.",folder,folder)) exit(-1)

// Build First Dialog (UserLevel)
DLGtg = DLGCreateDialog("Level",DLGitems)
DropMenu = DLGCreatePopup(DropMenuItems,1)
DropMenuItems.DLGAddPopupItemEntry("Novice user")
DropMenuItems.DLGAddPopupItemEntry("Experienced user")
DropMenuItems.DLGAddPopupItemEntry("Expert user")
DLGitems.DLGAddElement(DLGCreateLabel("Please select user level"))
DLGitems.DLGAddElement(DropMenu)

// Display First Dialog (UserLevel)
DLG1 = Alloc(UIframe).Init(DLGtg)
If (!DLG1.Pose()) exit(-1)

// Get the value (UserLevel)
DropMenu.DLGGetValue(userlevel)

// Build Sceond Dialog (Language)
DLGtg = DLGCreateDialog("Language",DLGitems)
DropMenu = DLGCreatePopup(DropMenuItems,1)
DropMenuItems.DLGAddPopupItemEntry("English")
DropMenuItems.DLGAddPopupItemEntry("German")
DLGitems.DLGAddElement(DLGCreateLabel("Please select command name language"))
DLGitems.DLGAddElement(DropMenu)

// Display Second Dialog (Language)
DLG1 = Alloc(UIframe).Init(DLGtg)
IF (!DLG1.Pose()) exit(-1)

// Get the value (Language)
DropMenu.DLGGetValue(language)

// Now install all your script. Modify this section manually to your needs.
AddLib2P("MyLibrary.s")
// The different "languages" lead to different installation trees
If (language==1)
        {
        If (UserLevel>0) AddS2P("script1.s","NOVICE Command",MenuName,"")
        If (UserLevel>1) AddS2P("script1.s","EXPERIENCED Command",MenuName,"")
        If (UserLevel>2) AddS2P("script1.s","EXPERT Command",MenuName,"")
        }
If (language==2)
        {
        If (UserLevel>0) AddS2P("script1.s","Anfänger",MenuName,"")
        If (UserLevel>1) AddS2P("script1.s","Benutzer",MenuName,"")
        If (UserLevel>2) AddS2P("script1.s","Experte",MenuName,"")
        }

// Install the functions necessary for version checking as library
AddScriptString2P(ScriptCompareVersion(),PackageName+"_check","","",1)
AddScriptString2P(ScriptOutputPackageInfo(),PackageName+"_info","","",1)
AddScriptString2P(ScriptUpdateButton(),"Plug-in info",MenuName,"",0)
```

How to use the template script

This third version of the template has quite a lot of source code. However, it does not need much modification for each new plug-in or for updating the plug-in.

To use this template on your own scripts you have to do the following:

- copy all scripts you want to a single folder on your hard disk
- copy this template to the same folder
- Note down the network path you want to use as a common "update-folder" for all scripts, e.g.: \\Server\Public\DM_Updates\
- Modify the template, updating the info at the top of the template:
  - string PackageName = "My_Package"
  - string MenuName = "Custom Menu Name"
  - number PackageVersion = 20071121
  - string GlobalPath = "\\Server\Public\DM_Updates\"
- Modify the template, building the (optional) user-level and language dialogues:
  - DropMenuItems.DLGAddPopupItemEntry("Novice user")
..etc..
  - DropMenuItems.DLGAddPopupItemEntry("English")
..etc..
- Modify the template, building the menu in the way you want, using individual lines of
  - AddLib2P("MyLibrary.s") // install script as library
  - AddS2P("My_script.s","MyScriptName",MenuName,"MySubMenuName") // install script as menu command
  - AddLine2P(MenuName,"MySubMenuName") // install a separation line in the menu
  - If (language==1)… // to built a structure for the individual language settings.
  - If (UserLevel>0)… // to include/exclude menu commands depending on the chose user-level on install.
- Start a "new" session with DigitalMicrograph after (temporarily) removing all unnecessary plug-ins from the plug-in folder: C:\Program Files\Gatan\DigitalMicrograph\Plugins\ (especially: If you are updating a plug-in, don't forget to remove the old one first!)
- Launch the modified template to install all scripts the way you want. (Menu commands should appear right away.) The plug-in is then stored as: C:\Program Files\Gatan\DigitalMicrograph\PlugIns\My_Package.gtk
- Copy the plug-in file to the chosen network path for distribution. \\Server\Public\DM_Updates\My_Package.gtk
- In the same folder, create an ASCII text file containing the information you want to display when the "plug-in info" command is executed: \\Server\Public\DM_Updates\My_Package_info.txt
- In the same folder, create a script file containing the single-command Exit(20071121), with the current version number of the package in format YYYYMMDD: \\Server\Public\DM_Updates\My_Package_v.s

If you follow these instructions, you will end up with a nicely structured way of managing your scripts as plug-ins shared with the wider community.