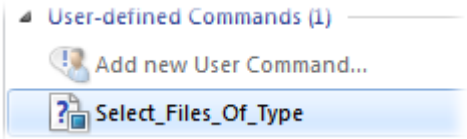
User-defined Commands

User-defined commands are pre-defined commands that you create yourself. Effectively it's like creating a toolbar button with your own function on it, but instead of the button living on a toolbar, it lives in this command list and other buttons or menus can refer to it by name.

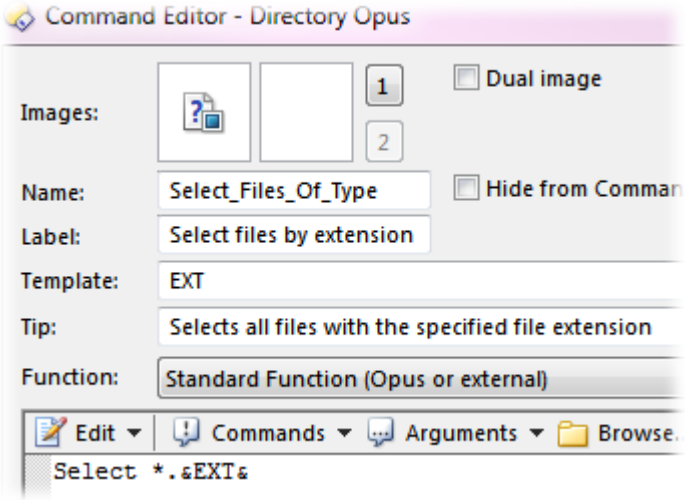
When you create a user command you give it a name (which must be unique and not clash with any of the Opus commands), and other buttons can then run your user command using its name. User commands can also accept parameters (arguments) on the command line, which lets you pass through parameters to external programs or scripts that your user command might invoke.



To create a new user command, double-click the *Add new User Command* item in the list, or right-click it or the *User-defined Commands* category header and choose **New** from the context menu. The context menu is the main way to perform other actions on user commands. If you right-click an existing user command, the commands in the context menu are:

- **Edit**: Edit the definition of the user command. You can also open the editor by double-clicking the command.
- **Duplicate**: Make a duplicate of the user command.
- **Copy**: Copy the command to the clipboard. You can paste it over an existing command (to replace its definition) or onto the *Add new* item to make a new copy of the command.
- **Export**: Export the command to a text file. This lets you share user commands with others. To import a user command use the **Import** command in the File menu.
- **Delete**: Delete the command.

The screenshot above shows an example of a user command in the list. As you can see, the name of the command is **Select_Files_Of_Type** (command names cannot contain spaces; any spaces are automatically converted to an underscore). Double-clicking that command to open the editor reveals the following configuration:



The user command editor is based on the standard function editor so we won't document fully how to use it here - instead see the Function Editor documentation for a full description. The principle elements of the user command are:

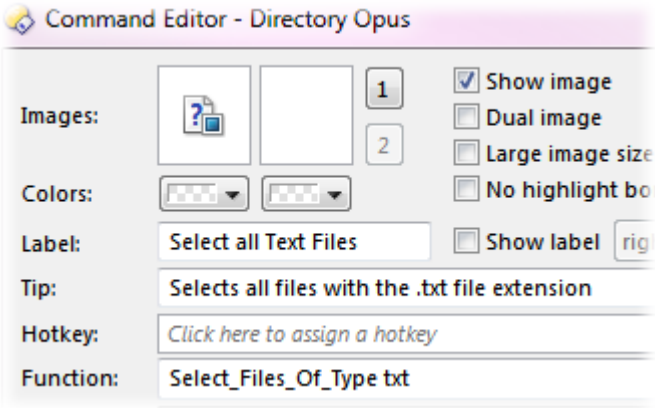
- **Images**: Defining an image for the user command sets its default image - the one that is used when it's dragged from the Customize dialog to a toolbar. This can be changed in the button after it's created.
- **Name**: The name of the user command. This is the name that other functions must use to invoke your command. If you change this and you have functions already set up to use your command, they'll stop working unless you update them with the new name.
- **Label**: A string that defines the default label for the button created when the command is dragged to a toolbar. This can be changed in the button after it's created.
- **Template**: This specifies the command line argument template for this command, if any is desired. See below for a discussion on the template.
- **Tip**: A string that defines the command's description. This is shown in the Customize editor when your command is selected, and is also used as the default tooltip when you drag the command to a toolbar to create a button.
- **Function**: Defines the function that this user command runs when it's invoked.
- **Hide from Command List**: If this option is selected, the user command will not be displayed in the drop-down command list shown in the command editor (the one displayed when you click the **Commands** drop-down in the above screenshot). This lets you create user commands that you can still use in buttons and hotkeys, but they won't clutter up the command list.

For Windows 7 users, the Jump List Preferences page lets you select individual user-defined commands that can be added to the Jump List menu. The **Label** and **Image** defined for the command will be shown in the Jump List.

When I created this user command I wanted a function that would select all files with a given file extension; therefore I needed to use the template field to specify a command line argument that is passed through to the command. This uses the same syntax as for internal Opus commands. In the above example, we only want a single string value. The argument has been given the name **EXT** in the template field. This is not the *value* of the argument (that doesn't exist until something uses this user command), but the *name* of the argument.

The function for the user command, **Select *.&EXT&**, calls the internal **Select** command, passing it a wildcard string that is built from the supplied **EXT** parameter. The argument name is supplied in the function definition surrounded by ampersands (**&**) which indicates to Opus that the value of that argument is to be inserted in the command line.

So how do we actually use this user command? Well, because it requires a command line argument to do anything, we need to put it on a button or menu (or hotkey) and then edit the function to supply the desired extension. The following button definition would do this; it will select all **.txt** files in the current file display.



As you can see, we have edited the label and tooltip (description) to one more suited to the actual button we created. The function defined for this button calls our new **Select_Files_Of_Type** user command, passing it the argument **txt**. The user command takes the argument and passes it on the command line as described above to the **Select** command, which then performs the desired action by selecting all files that end with a **.txt** extension.

In the above example, **EXT** is a string argument and so the value supplied is passed through unchanged to the user command, but for boolean options the behaviour is different. For example, consider the following user command:

```
Name:       ExampleCmd
Template:   EXAMPLE1/S,EXAMPLE2/O
Function:   C:\DummyProgram.exe &EXAMPLE1& &EXAMPLE2&
```

The **ExampleCmd** user command will run the *C:\DummyProgram.exe* program when it's invoked. Its template has two boolean options, **EXAMPLE1** (a straight switch argument, either on or off), and **EXAMPLE2** (an option switch argument, which can be off, on or have a string value supplied). By default, a switch argument will insert the value **1** when it is set, and **0** when it isn't. For example, the following use of the user command would produce the following command line for *DummyProgram*:

```
Command:    ExampleCmd EXAMPLE1
Runs:       C:\DummyProgram.exe 1 0
```

However, you can use the **&..&** insert to specify the actual strings that are passed through for that argument. For example:

```
Function:   C:\DummyProgram.exe &EXAMPLE1:yes:no& &EXAMPLE2:one:two&
Command:    ExampleCmd EXAMPLE2
```

```
Runs:       C:\DummyProgram.exe no one
```

For option switch (**/O**) arguments this only applies if the argument has been used as a switch, and not to provide a string value - if a string value is supplied instead, it is passed through unchanged. For example:

```
Command:    ExampleCmd EXAMPLE1 EXAMPLE2=test
Runs:       C:\DummyProgram.exe yes test
```

Your template can also specify a list of values for an option switch which will then be shown in the drop-down list in the advanced command editor, making it easy to pick the value for the argument. If you use this, you can also specify a default value, which will be passed through if a value is not given for the option. Consider the following template and the two example uses:

```
Name:       ExampleCmd
Template:   EXAMPLE/O[<default>,one,two,three]
Function:   C:\DummyProgram.exe &EXAMPLE&

Command:    ExampleCmd EXAMPLE
Runs:       C:\DummyProgram.exe default

Command:    ExampleCmd EXAMPLE=two
Runs:       C:\DummyProgram.exe two
```