# Before you begin

Before recording the UI steps for automation, you should be familiar with the manual test case, and ensure the steps can actually be performed as described. Try to have the minimal number of steps that are distinct and unambiguous, and will work in all situations. Then you can use the RPF recorder to record those steps on a Pseudo-loc build.

RPF, “Record and Playback Framework” is a wrapper around different UI Automation modules. The key ways we interact with RPF are:

Recorder, which logs user actions and automatically generates code to repeat those actions.

Playback, a collection of functions that take QueryIDs and ResKeys to perform actions.

The main things preventing recorded code from playing back on any windows machine are: Localization, Platform, and Edition differences.

Recording in generally performed on Pseudo-Localized builds, as they guarantee that each string is unique. Making the code work on different languages is mostly automatically taken care of.

Platform (i386 vs. amd64) differences crop up sometimes, there are SKUDIFF\_UTIL functions that handle most of the issues automatically, but they require that your resKeys be formed in a particular way which is not the default.

The automatically generated code, produced from a single SKU is not SKU portable, and does not log results in the way we need, so we generally cut and paste small bits of the generated code into our code.

# Installing RPF

For Vista SP1 we are using RPF 11.7, Windows 7 will likely use a newer version, so be careful when relying on documentation that may not apply.

Generally, we allow WTT to install the core RPF files on the test client machines automatically. This is triggered by the constraint that is defined for all of our test jobs of “WindowsShell-RPF-LH” which causes job #6764 to be run.

The files needed to build tests that use RPF come in the SHELLTEST enlistment of the NT codebase, which is also where our code is built. RPF is implemented as a COM DLL, so linkage occurs at runtime.

The recorder depends on parts of the client, but once the client has been installed on a machine you simply need to copy the matching version of the recorder folder and execute it.

# Recording

The first stage is to run the RPF recorder and record your UI interaction steps, see the RPF documentation for detailed instructions and troubleshooting.

After recording, you can choose to Generate code automatically, For Vista use ‘Native C’, for Windows 7, ‘C#’ is preferred. Be sure to choose to Globalize the code on the second tab of the Gerenate Code Dialog, otherwise you won’t get ResKeys.

RPF Will notify you that the code is available on the clipboard, which you can paste into Notepad, or another editor.

You now have a source file, which you can pick pieces from for use in our test code.

Be careful not to copy any non-ascii characters to your source code, as that can cause build issues.

Ideally, you can use a copy of an existing test case, or a template if available, and copy the constants defined in the generated code to your new test case as described below.

# Resource Keys

A ResKey specifies how and where on a test machine to pull a localized string from, primarily for use by QueryIDs.

ResKeys are constant strings, as examples:

const \_\_declspec(selectany) wchar\_t\* resKeyStart = L"RKB1[];Start;Win32String;%windir%\\system32\\oleaccrc.dll;271";

const \_\_declspec(selectany) wchar\_t\* resKeyStartWindow = L"RKB1[];Start;Win32String;%windir%\\explorer.exe;595";

RPF reskeys default to specifing a process name (inside the []’s) and a relative path (“$.”); this will not work many entry point tests, since the process isn’t running until it is entered. To correct this, we remove the process name from within the []’s and specify a full variable path to the file.

For example:

"RKB1[explorer];More;Win32String;$.\\explorer.exe;7026"

Becomes:

"RKB1[];More;Win32String;%windir%\\explorer.exe;7026"

In the common header I used:

#define RK const \_\_declspec(selectany) wchar\_t\*

To make declaring a reskey easier; “\_\_declspec(selectany)” is needed to allow the header to be referanced more than once for the same build target.

The existing value of the resKey, between the first and second delimiter is usually poorly de- PseudoLoc’d by RPF, so I often correct the spelling and tidy the string. The value given is generally ignored by RPF when using the resKey, so it appears to only be of documentation value.

Often, a resKey you just recorded will already exist in the common headers, the easiest way to determine that is to search for the portion of the string that contains the filename and resource number, such as “shell32.dll;30579”

Some custom resource extractors have been written for specific test cases. SKUDIFF\_UTIL has custom extraction functions for drive labels, optional Windows Features, and game explorer .gcf files.

I think a good practice going forward would be to keep the five character PseudoLoc hash, to uniquely identify the string, and link the PS UI to the Resource key, that way a test developer can just search for the 5 character PseudoLoc hash to check if a string has already been found.

# QueryIDs

A QueryID is a chain of attributes of UI elements starting from the Desktop, used to target a specific element. It is **NOT** a series of controls to manipulate, but a series of parent-child windows to find a particular window.

The challenge of a well formed QueryID is to write it so that it always finds the element you want in different scenarios, but not any other element. More information on QueryIDs is available at [\\rpfbuilds\Release\Generation1\Version117(RC1)\Documentation\QueryId 11.doc](file:///\\rpfbuilds\Release\Generation1\Version117(RC1)\Documentation\QueryId%252011.doc)

Currently, we define a QueryID as a preprocessor macro, for example:

#define START\_BUTTON L";Name = '{0}' && ClassName = 'Button';Name = '{1}' && Role = 'push button'", resKeyStartWindow, resKeyStart

From RPF generated code the QueryIDs may be in the body of the code as literal string, and will need to be copied out and given names

Each element is separated by whatever the first character in the string is; in this case, and the default, a semicolon: ‘;’

Attributes of Elements are joined by the C logical-and operator ‘&&’, other operators are not supported in RPF 11.7 to my knowledge; newer versions may have other interesting operators.

We usually remove any occurrences of ‘[VisibleOnly]’ from the QueryIDs, as it makes the tests more reliable.

If you previously found that the resKeys used by your Query already existed, then you can check if the QueryID that uses it already exists by searching for the constant name of the pre-existing reskey.

During playback, RPF may attempt to automatically ‘fix’ your QueryIDs if it doesn’t find the specified item. It does this by removing intermediate elements, for example:

L";Name => '{0}' && ClassName => '#32770';ControlId = '2';Role = 'push button'", resKeyDialogName

Might be converted to:

L";Name => '{0}' && ClassName => '#32770';Role = 'push button'", resKeyDialogName

Causing RPF to click some other arbitrary button, if Control ‘2’ is not found; this can become VERY confusing. This behavior can be turned off… only in newer versions of RPF. The best way to deal with this is the ensure that your first and final Elements are sufficient to uniquely identify the target window. In the above example, Control 2 would be a ‘Cancel’ button (generally, 1 is ‘OK’, 2 is ‘Cancel’), you could target it better by using something like:

L";Name => '{0}' && ClassName => '#32770';Role = 'push button' && Name = ‘{1}’", resKeyDialogName, resKeyCancel

My suggestion for a future design would be to store the ResKeys and QIDs in a data file so that they can be:

A) read from any language, managed or not, making it easy to move to newer frameworks.

B) modified to adjust for localization changes without changing the code, avoiding check-in hassles.

C) named/labeled so that the test code and logging can be simpler.

# Coding:

If some of the QueryIDs and ResKey already existed, there may already be code to do part of the task you recorded. Native C++ functions exist in SKUDIFF\_UTIL for opening the Start Menu, the All Programs menu, most standard Control Panels, and the Welcome Center. See “SKUDiff Shared Library.docx” for more information.

## General test function flow

### Pass if test blocked for a well known reason.

For example, if the graphics card isn’t sufficient, or the feature is co-differentiated by something other than SKU, such as language (Branding, MUI), or being domain joined (Parental Controls).

Log the reason, and report a pass. The ‘LogKnownBug’ function might be useful here.

While a ‘false pass’ is a risk, if the test runs report to many false failures all failures may end up being disregarded.

### Set expected behavior based on SKU

Usually a case/select mechanism is used

### Navigate to feature location.

This can mostly be done with the Common UI functions from SKUDIFF\_UTIL, such as ‘OpenWelcomeCenter’ (see “Common UI action functions” below)

### Check for existence.

Preferably use a function like ‘IsAvailable’ for the target UI element, instead of depending on a ‘failure’ from RPF, since RPF may automatically attempt to ‘correct’ the failing Query. You can also examine attributes of a UI element via ScreenElement methods.

### Fail if test blocked for unknown reason.

If something unexpected happens that keeps your test from checking its target, (for example, you are checking the state of an element, but you can’t find the element in the first place) report a Fail.

### Check if Expected behavior matches Actual behavior.

Usually, we simply check if Actual == Expected. If they differ we fail, if they match we succeed.

### Cleanup

Close the windows you open, restore changed system settings, etc.

# RPF Wrapper Functions

Defined in SKUDIFF\_UTIL.h, and defined in RPFWrapper.cpp, these functions are designed to take the same parameters and return (on 32 bit systems) the same results are the equivalent normal RPF function, that is, the function with the same name except for the ‘WOW’ at the beginning. The main difference is they will automatically attempt to adjust a resKey that contains “%programfiles%” or “%windir%” to the 32-on-64 equivalent automatically. However, they will only do this if the variable path portion is specified in lower case; this could be considered a bug, but I decided to leave it as a way to bypass the redirection if required.

HRESULT WOWFromPartialQueryId (\_\_in PCWSTR pszQueryId, ... );

ScreenElement\* WOWScreenElementFromPartialQueryId (\_\_in PCWSTR pszQueryId, ... );

bool WOWIsAvailable (\_\_in unsigned int nTimeOut, \_\_in PCWSTR pszQueryId, ...);

HRESULT WOWGlobalizeQueryId (\_\_out\_ecount(cchGlobalizedQuery) PWSTR GlobalizedQuery, \_\_in unsigned int cchGlobalizedQuery, PCWSTR pszQueryId, ... );

# Common UI action functions

Also declared in SKUDIFF\_UTIL.h, but defined in CommonUI.cpp, They generally perform the action that their name implies, “OpenSpecificControlPanel” is unique in accepting parameters, it can accept a single QueryId for a ‘classic’ control panel icon, or a pair of QueryIds for a control panel section and item link. See ‘SKUDiffLibrary.docx’ for more information.

HRESULT OpenStartMenu();

HRESULT OpenAllPrograms();

HRESULT OpenWelcomeCenter();

HRESULT OpenControlPanelClassicView();

HRESULT OpenControlPanelHomeView();

HRESULT OpenSpecificControlPanel(\_\_in PCWSTR pszQueryId, ...);