# SKU Diff coding tips

## Consistent State

Try to return the system to the state it was in before the test function started, close any windows you opened, restore any settings, etc.

## Don’t depend on the default state:

Other tests may not have been able to restore the system state, so try and ensure that commonly changed settings don’t break your test, such as the Start menu or Control Panel being in classic mode.

## Atomic:

Each test function ideally tests one, and only one thing, since the main output is a PASS or FAIL. One of the limitations of the current test framework, is that once a test begins, it cannot become N/A, it must PASS or FAIL.

## BOOL vs. bool (and HRESULT)

‘bool’ (lowercase) is a built-in C++ type that can have a value of ‘true’ or ‘false’.

BOOL is a Microsoft specific type, created before bool was added to C++, internally it is an integer. It has two associated constants, ‘TRUE’ and ‘FALSE’, which are equal to 1 and 0 respectively.

A problem arises if you treat a non-zero/one value as a BOOL, for example, “if(2)” is true, while “if(2==TRUE)” is false.

A ‘HRESULT’ is another type that is actually an int; however for an HRESULT, zero means success, anything other than zero is a failure code; in other words, Success is False, Failure is True.

When testing an HRESULT always use the ‘SUCCEEDED()’ or ‘FAILED()’ macros and when testing a BOOL use ‘== TRUE’ or ‘== FALSE’ to clarify the returned type expected, otherwise it may not be obvious to someone working on the code later what type the expected result was.

When writing a function that performs an action that is possible to fail, try to return an HRESULT, Avoid replacing a specific HRESULT from an inner function with a generic failure, pass the inner result along.

If the function name implies a question, then it may be ok to return a BOOL/bool, as the code calling the function should have an ‘else’ clause.

## Logging Functions, interesting C++ preprocessor features, and ‘\_\_FUNCTION\_\_’

Throughout the code you’ll see things like “\_\_FUNCTION\_\_” , “LOG\_FAILED\_HR” they are pre-compiler directives that get replaced at build time.

The details are too much to go into here, but understanding the compiler features used to build them is useful.

“\_\_FUNCTION\_\_” is A C++ macro that becomes a string literal of the current function name, “\_\_FUNCTIONW\_\_” is the Unicode version. There is also “\_\_FILE\_\_” and “\_\_LINE\_\_”, an example of usage:

LogTestInfoW(\_\_FUNCTIONW\_\_, L"Have to re-open All Programs menu.");

‘#’ turns the following macro parameter into a literal string, I used it in the UTIL functions to make logging easier:

#define LOGRESKEY(reskey) (LogResKeyValue(L#reskey,reskey))

HRESULT LogResKeyValue(\_\_in const wchar\_t\* szResourceKeyName,\_\_in const wchar\_t\* szResourceKey)

These features are combined in the LOG\_FAIL\_HR family of macros, to provide detailed information in the test log with minimal effort, As an example, when this line fails to find Games on the Start Menu:

HRESULT hr = LOG\_FAILED\_HR(FromPartialQueryId(STARTMENU\_GAMES));

LOG\_FAILED\_HR logs the error code, contents of the failing expression, and the location in the source of the error.

Got error 0x80004005 (Unbekannter Fehler) from FromPartialQueryId(STARTMENU\_GAMES) in OpenGamesExplorer at games\_bvt\_entrypoints.cpp:205

## Be aware of multi-value macros.

RPF QueryIDs are used in the current code as precompiler macros, so this note applies to them. But for a simpler example;

“CLICKABLE\_POINT” is a Macro for “-1, -1”, which RPF interprets as ‘anywhere’, a function that takes “CLICKABLE\_POINT” as a parameter actually takes two parameters.

QueryIDs can have 0 or more reskeys, so functions that take them take a variable number of parameters. When I wrote a function that needed multiple QueryIDs as parameters, I had to convert each into a single parameter. Fortunetly, the conversion functionality mostly exists in RPF already in the form of the “GlobalizeQueryId()” function, which turns a QueryID into one string with the reskeys inserted.

wchar\_t Section[1024];

wchar\_t Link[1024];

if (SUCCEEDED(GlobalizeQueryId(Section, 1024, CPL\_LINK\_PROGRAMS)) &&

SUCCEEDED(GlobalizeQueryId(Link, 1024, CPL\_LINK\_PROGRAMSFEATURES)) &&

OpenSpecificHomeControlPanel(Section, Link) == TRUE)

This if statement also takes advantage of the C++ rule that when ‘&&’ing multiple expressions, if the left-hand portion is false, it does not bother to execute the right hand portion; so that if eighter ‘GlobalizeQueryId’ call fails, the ‘OpenSpecificHomeControlPanel’ function is not called.

I recoded the functions to take QueryIDs that are not pre-golabized, but this is still something to be aware of. Newer versions of RPF have a QueryID class defined, which is unfortunetly not available for the Vista test sources.

## 64 bit systems

~~I recently wrote a number of functions to adapt existing 32 bit working test code to run on 64 bit systems; it’s important to note that the test code is not being built in 64 bit, it’s just 32 bit code that manipulates the UI of applications that may be 64 bit sometimes. Since most of our tests are using UI manipulation, the main effect this has on our test code it that the binaries that we pull resources from may be in different directories than expected.~~ **Still needed, but for a different reason**

To deal with this, I wrote wrappers around a few RPF functions, such as “WOWFromPartialQueryId” which re-implements some globalization code so that it searches multiple paths for the binary with resources; fortunatly, even when re-compiled as 64 bit, the resources for a given program keep the same type and index numbers. Currently this code depends on “%programfiles%” and “%system32%” being used in the reskeys to specify the path; which should be the case for most binaries.

## Header Files used:

#include <windows.h> //Common Windows types, in public sdk

#include "rpf\_constants.h" // SHelltest-wide RPF constants, in Shelltest\inc

#include "screenelement.h" //RPF Shelltest\tools\rpf\rpfplaybacknamespace

#include "LoggingFunctions.h" //SUCCEEDED, FAILED, etc defined here Shelltest\inc

#include "rpf\_skudiff.h" // SKU Diff RPF Constants Shelltest\inc

#include "SKUDIFF\_UTIL.h" // SKU Diff utility functions Shelltest\inc

## InitTest / UnInitTest

InitTest is called by MASH before running test functions. UnInitTest is called after the set of test functions.

A MASH .SWT file can specify test functions that reside in different .DLLs, however I don’t know how it determines which set of functions it will attempt to call, since our current .SWT files only use 1 test .DLL each. This should only become an issue if the test functions are .SWT files change alignment.

## GetProductInfo

A windows API function, only available in Vista+, which is why our code won’t build under CRESCENT\_QFE. We call it specifying Windows Vista, if tests are run under a future version of Windows, this API will supposedly map the current edition to the best matching Vista edition.

## Control Panel

The Control panel has multiple modes, and test automation needs to take into consideration that it may not be in the mode you want when you open it. Also, the “Control Panel” item on the Start menu can appear in many forms, or not at all.

The control panel section “User Accounts (and family settings)” will change name, not just on different SKU’s, but also depends on Domain status.

The same item can appear in multiple categories, for example Windows Firewall applies to both Networking and Security.

## Boundary Machines

At Microsoft, due to security policy, most machines joined to the Redmond domain will refuse to communicate with non-domain join machines. Only special servers called “Boundary” machines can communicate with both.

Since Home and Starter SKUs cannot join a Domain, any files they need to download need to be put on a Boundary machine. (Packmule and Shelltest, and now WEXFS are existing boundary machines, Anything new should probably go on WEXFS, however, Shelltest is being used for Vista SP1 purposes.)

## Multilanguage User Interface

MUI is only available with an English base system. This also excludes ‘Pseudo-Loc’ so that while it is a key Internationalization feature, it’s not available on international builds.

## Tests should fail by default

Any variables in test code should be initialized to a state that will indicate a failure if the following code fails; however the current code will ‘Pass’ is the tests is blocked for a known reason, such as detectable insufficient hardware; this is to prevent excessive false failures, which may lead to the test code being discarded, as well as to highlight real bugs/product failures.

# Adding a Test Case to your local build

In this section, when referring to a path, “…” is the location of your Source Depot enlistment Root, and “???” is the name of your test case.

## Create the directory

Currently I am creating test cases in folders under “…\testsrc\shelltest\testareas\SKUTest\???”

The foldername must not contain spaces, and should not be too long, as there is a total path size limit; but avoid TLA’s (three letter acronyms).

## Create the basic test files

The essential files are:

the .cpp file, containing the actual test source

the .def file, specifying the functions exported from the .DLL to be built.

the .swt file, specifying the test functions for MASH to call when executing

the sources files, specifying the files needed to build the test .DLL, as well as the .swt file in the miscfiles section

I recommend copy-pasting an existing test folder as a starting point.

## Modify the ‘dirs’ file

“…\testsrc\shelltest\testareas\SKUTest\dirs”

The ‘dirs’ file contains the list of subdirectories for the build process (bcz in a Razzle window) to process; related to the ‘sources’ file. The ‘\’ character is not a separator; it is used for line continuation. Adding blank line in the middle will break the file, as it considered to be one long line by the build.

## Modify the ‘placefile.tst’ file

“…\testsrc\shelltest\placefil.tst”

To find our section, I usually start with a text search for ‘SKUTest’, which should take you to the SKUTest section.

Add the .DLL and .SWT files for your testcase.

# How does my test code get onto the test machine?

The normal way, for an actual build, is that WTT copies the required files from the testbinroot, which exists on the same share as a sibling folder to the OS install point.

However, since we have not been able to check out code into the NT codebase, we use a network share. This share needs to be on a ‘Boundary Machine’ so that it can be accessed if the test machine is not joined to a domain.

You can re-direct where WTT copies test code from by running this command on the client machine.

**wttcmd /configreg /add /value:WTT\TestBinRoot /data:\\server\share\path\x86fretest\bin**

The tests currently expect the debug symbol files for the test files to be in a ‘sibling’ directory of the TestBinRoot, accessed by ‘[TestBinRoot]\..\symbols.pri’ so the directory you need to create should be within another directory of yours (‘x86fretest’ in the example above’), I have generally switched the cases in WTT to not fail if the symbols are not found, since they don’t affect to actually running of the cases.

All of our tests require the setting of a user for the job to use when accessing the network.

**wttcmd /addlogicaluser /localname:LLU\_Default /user:igloolab /domain:redmond /password:xxxxxxxxxx**

The account you use for LLU\_Default needs access to the share you specified for TestBinRoot.

After changing any WTT client setting, it’s strongly recommended to restart the WTT Service:

**sc stop wttsvc**

then

**sc start wttsvc**