Application Report
SPRAAB7 – October 2005

Automated Regression Tests and Measurements with the CCStudio Scripting Utility

Vincent Wan, Ki-Soo Lee *SDO Applications*

ABSTRACT

The Code Composer Studio Scripting Utility is an interface that provides access to CCStudio functionality from either Perl or COM-based languages.

Using the CCStudio Scripting Utility, a simple user script can automate many common actions within the CCStudio IDE. More powerful scripts that take advantage of the functionality provided by scripting languages, such as Perl or Visual Basic, can be used with the Scripting Utility to create a test bench that fully automates regression testing of an application. Important measurement data such as profiling and code coverage statistics can also be attained. This application note introduces CCScripting and shows an example that automates profiling of an application framework (Reference Frameworks Level 3).

Contents

Figures

Tables

1 Introduction

As embedded development continues to grow in both scope and complexity, so do the demands on unit and system testing. Automating such testing is a crucial part of the development cycle; it can substantially decrease time to market.

The Code Composer Studio Scripting Utility allows you to use familiar scripting languages such as Perl or VBA to automate testing and validation of applications. It exposes many useful APIs that perform common actions such as launching CCStudio, opening projects, loading programs, setting breakpoints, and so on. The Scripting Utility is valuable to anyone who wants to use CCStudio to run tests in batch mode and/or write regression tests in Perl or a COM-based language.

This application note briefly introduces the CCStudio Scripting Utility and gives an example that automatically profiles an application framework (Reference Frameworks Level 3). The examples are written in JScript, an interpreted, object-based scripting language that is Microsoft's implementation of the ECMA 262 language specification. For information on JScript, see the MSDN Library.

2 System Requirements

The software prerequisites to run the example provided with this application note are:

- Code Composer Studio v3.1 or greater.
- CCStudio Scripting Utility v1.5 or greater. Available for download through the CCStudio Update Advisor.
- Windows Script Host (WSH) 5.6 or greater.

3 CCStudio Scripting Utility

The CCStudio Scripting Utility provides easy batch-mode access to CCStudio debugging and testing functionality through a simple API interface. You can call these APIs from either Perl or a COM-based language such as Visual Basic and JScript. While Visual Basic is the COM-based scripting language supported by the Scripting Utility in terms of examples and documentation, the Scripting Utility works with any COM-based scripting language.

3.1 How it Works

The Scripting Utility exposes COM and Perl interfaces to CCStudio APIs. There are over 50 such APIs. User scripts call the interface APIs to access the CCStudio APIs and automate actions in CCStudio as shown in Figure 1. The Scripting API Reference (Reference 1) document contains a full list of the APIs available and information on the basic functionality of the Scripting Utility.

All the Scripting APIs have fully synchronous behavior, meaning that the API waits until its action is completed and the status (along with any other information specific to the API) is returned.

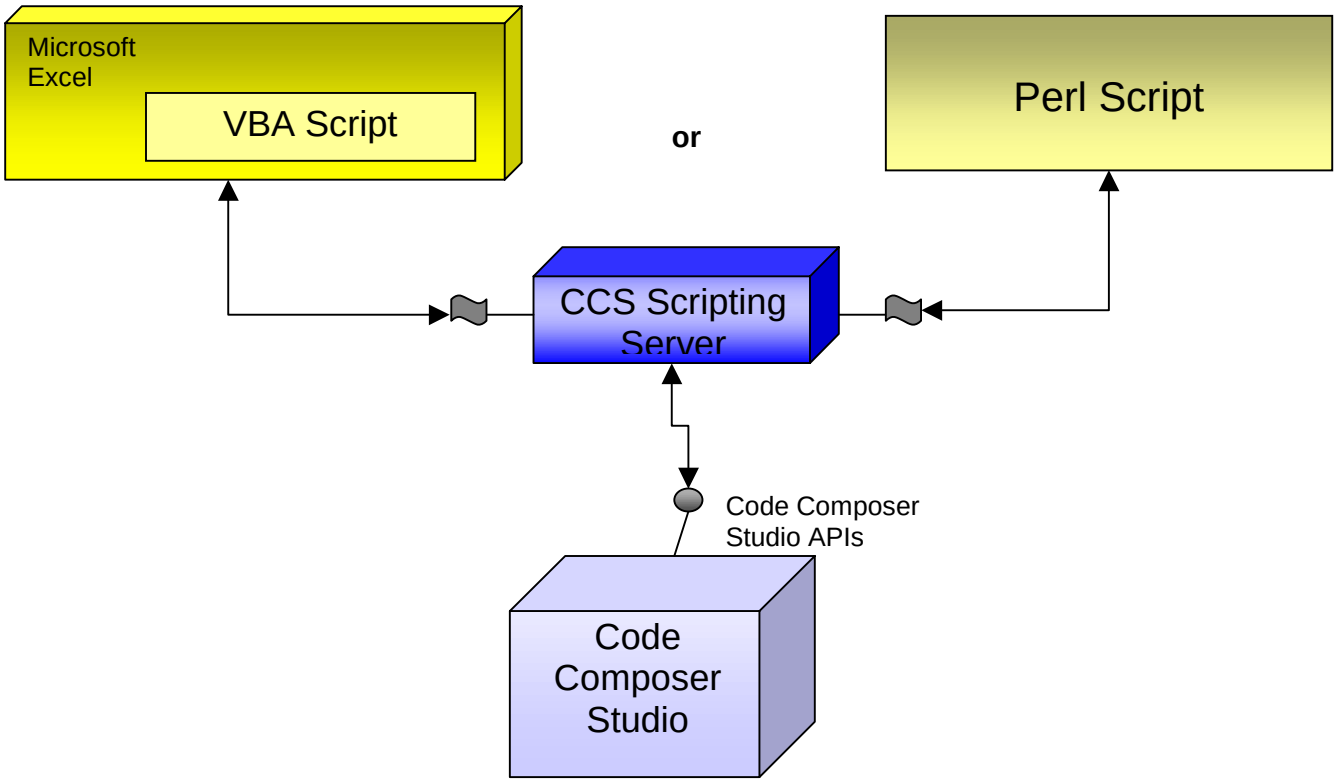


Figure 1. Flowchart of CCStudio Scripting Utility's Behavior

3.2 CCStudio Scripting Utility vs. GEL

People familiar with CCStudio are also most likely familiar with GEL (General Extension Language) and use it to automate some tasks within their CCStudio environment. So why use CCStudio Scripting instead of GEL? GEL has several limitations that prevent it from being an ideal solution for automated testing:

- It can only be run within the CCStudio IDE. Any GEL actions cannot be performed outside of the IDE. It cannot open or close CCStudio.
- GEL is not fully synchronous. Some of the *built-in* GEL calls are asynchronous, making it difficult to create long, complex scripts where current actions depend on prior actions being

fully completed. For example, a large program (.out file) may not be fully loaded (via GEL_Load()) by the time it starts execution (via GEL_Run()).

- GEL scripts apply to the control window (a single instance of the CCStudio IDE debug window) it is run from (with a few exceptions). This limitation is noticeable during heterogeneous debugging where there can be more than one CCStudio control window.

The Scripting Utility addresses all of these issues. Scripts can run outside the CCStudio IDE and can open and close CCStudio. All of the APIs are fully synchronous, and the script can access multiple control windows that are open.

Despite the advantages of the Scripting Utility, there are still benefits of using GEL. Some built-in GEL calls offer capabilities beyond those the Scripting API possesses on its own. To remedy this, a special Scripting API called TargetEvalExpression() is provided. This API accepts any recognized GEL call as its parameter. This allows the script to take advantage of any GEL functionality not exposed via the Scripting APIs.

The Scripting Utility does not replace GEL usage. Rather, it supports GEL usage for automation. Using CCStudio Scripting and GEL together provides even greater functionality and flexibility.

3.3 CCStudio Scripting Limitations

CCStudio Scripting has the following limitations:

- Not all CCStudio plug-ins are accessible via scripting. If a plug-in does not expose its APIs, then the Scripting Utility does not have access to it. One example is the DSP/BIOS Real-Time Analysis tools in CCStudio.
- Not all versions of Perl are supported. Scripting Utility v1.5 supports ActivePerl 5.6 or 5.8, depending on the ActivePerl version you specified during installation. Once you have selected a version of ActivePerl support, you must uninstall and then reinstall the Scripting Utility to use the other version of ActivePerl. To avoid this, make sure you select the correct version of ActivePerl is when prompted.
- If you use a scripting language that supports early binding (such as Perl or VBA), your script can refer to the built-in constants described in the CCStudio Scripting documentation (for example, ISA_C64 and VERBOSE_ALL). If you use a scripting language that uses late binding (such as JScript), you must define the constants in the script by looking up the values in the documentation. For convenience, the file ccs_scripting_constants.js has been provided with this application note. Your script can include it to automatically import these constants when using JScript. A similar approach can be taken for other scripting languages that support only late binding.

  NOTE: Both Visual Basic and VBA (Visual Basic for Applications) support *both* early and late binding. VBScript, a lightweight version of the Visual Basic language, supports only late binding.
- While TargetEvalExpression() is a synchronous call in that it waits for a response, when the response is returned depends upon the GEL call passed in. If the GEL call is asynchronous, TargetEvalExpression() receives an immediate response that the GEL call was evaluated and returned, even though the action started by the GEL call may not be complete. This

applies to any asynchronous built-in GEL call. See Appendix A for a listing of GEL commands and their synchronicity.

The following example shows one problem that could potentially arise when using asynchronous GEL commands in scripting:

```
ccs.TargetReset();
ccs.TargetEvalExpression('GEL_Load("app.out")' );
ccs.TargetRun();
```

The above code may not work. Since GEL_Load() is asynchronous, TargetEvalExpression() may return before the program has finished loading, causing the target to start running early. The correct way method is to use the Scripting ProgramLoad() API as follows:

```
ccs.TargetReset();
ccs.ProgramLoad(“app.out”);
ccs.TargetRun();
```

This API ensures that the program is loaded before returning. In general, it is preferable to use Scripting APIs instead of equivalent GEL commands.

3.4 What’s New in CCStudio Scripting v1.5

In addition to numerous bug fixes and minor enhancements, version 1.5 provides the following:

- Added support for ActivePerl 5.8. During installation, you are prompted for which version of ActivePerl you want to be supported by the Scripting Utility.
- Additional profiling support through a new API called ExportProfileData(). This API exports collected profiling data to an *.csv file or Microsoft Excel spreadsheet format.

NOTE: CCStudio Scripting v1.5 does not support CCStudio versions prior to CCStudio v2.40

4 Example: Automation of RF3 Profiling With CCStudio Scripting Utility

This section describes how to automate profiling of Reference Frameworks Level 3 (RF3) using the CCStudio Scripting Utility. Cycles are collected with the multi-event profiler in the Analysis Toolkit (ATK) available in CCStudio 3.1. In order to use this tool, we first ported the application to work on the ‘C6416 device cycle accurate simulator (See Reference 2 for details). The simulator attempts to imitate the behavior on a ‘C6416 DSK using pin and port connect.

By using the script described in this section, you can automate running an RF3-based audio application and profiling the functions in the application.

4.1 Overview of the Automation

This example uses JScript combined with the Windows Scripting Host. This is a logical choice for a developer who uses DSP/BIOS and is familiar with its textual configuration tool (Tconf) which is JavaScript-based, a close variant of JScript.

Figure 2 shows the automation flow:

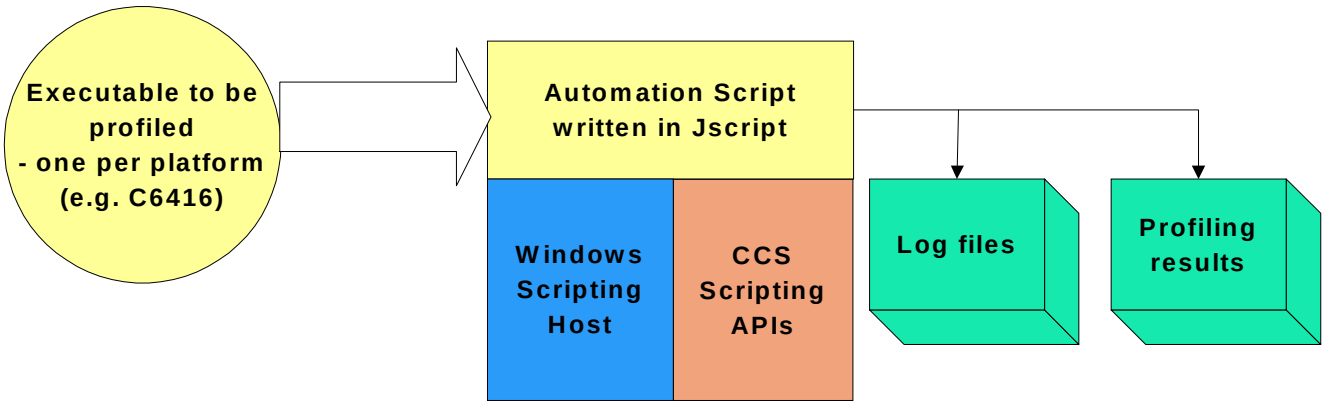


Figure 2. Flowchart of RF3 Profiling Test Bench

The goal for this example is to load and run the executable for the equivalent of the duration of the processing of one audio frame. The ATK can then be used to provide a breakdown of the cycles spent in each function called. A log file indicating the status of each automation call and an Excel spreadsheet containing the profiling results are produced by each run of the script.

To run the test script, we created a Windows script file called test.wsf as follows:

```
<Job id="test">
/* Includes */
<script language="JScript" src="../include/ccs_scripting_constants.js"/>
<script language="JScript" src="../include/ATKtprof2xls.js"/>
<script language="JScript" src="../include/PrintWScriptArgs.js"/>
/* Main script */
<script language="JScript" src="test.js"/>
</Job>
```

This allows us to include three files when running the script test.js that does the profiling. These files define constants and enumerations that are used as part of Scripting, and two functions that are used as part of the script: ATKtprof2xls() and PrintWScriptArgs(). As explained in Section 3.3, the ccs_scripting_constants.js file is included to overcome the limitation that prevents scripting languages that uses late binding (such as JScript and VBScript) from directly accessing enumerations exported by scripting.

Run the script, from a command prompt, using this syntax:

```
cscript test.wsf <list of arguments for test.js script>
```

For your convenience, we provide a batch file called runtest.bat to show you an example of the arguments that are passed to the script. See Section 4.3 for details.

4.2 Contents of the Companion Code

The companion zip file for this application note contains the files used for the example. Unzip the file into a directory named my_test_folder, as shown in the Figure 3:

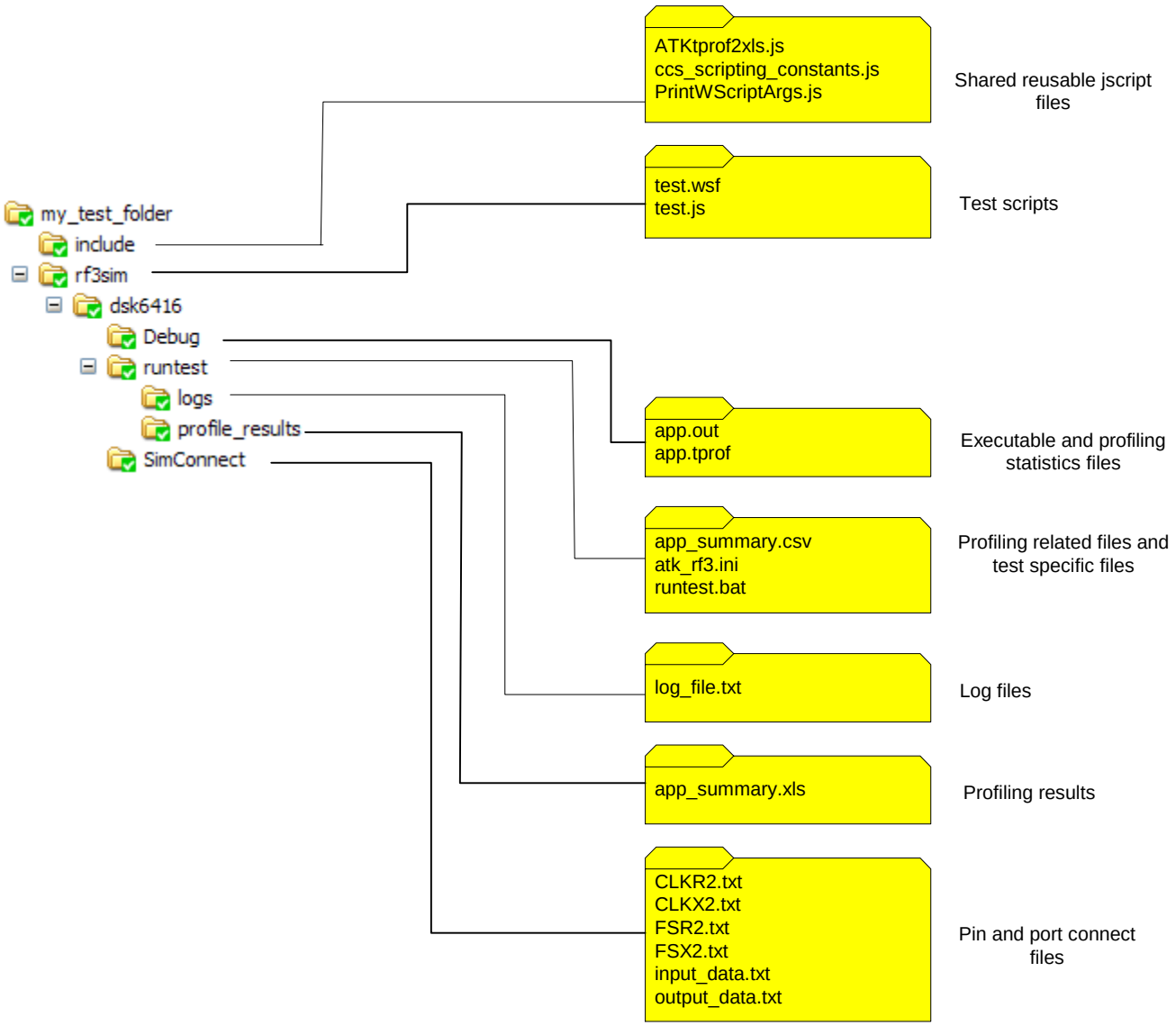


Figure 3. Example Test Directory Structure

The folders under my_test_folder are as follows:

- **include.** Contains shared JScript files with reusable functions and constants.
- **rf3sim.** Contains the test scripts for the project. This folder holds subfolders that contain files specific to different platforms.

Each platform folder contains the following subfolders:

- **debug.** Contains the executable and the raw profiling statistics (.tprof) file is created by the ATK when the application is run.
- **runtest.** Contains the batch file for running the test script along with the profile configuration file and intermediate files generated by the ATK during the spreadsheet generation process.

  - **logs.** Folder used to store log file from each run of the test script.
  - **profile_results.** Folder used to store result spreadsheets generated by the ATK.
- **simconnect.** Contains files necessary to use pin and port connect during simulation.

4.3 Running the Script

To run the example, follow these steps:

1. Configure the 'C6416 Device Cycle Accurate simulator in CCStudio Setup as per Reference 2. It needs to be the only platform configured in order to run the example successfully.
2. Modify the basePath and ccsInstallDir arguments in the runtest.bat file located in the runtest directory to match your specific setup. The next section has more information about these parameters.
3. When you are done, enter "runtest" at the command prompt to run the script.

4.4 Understanding the Script

This section examines the script file test.js piece by piece. Despite the fact that the script is written in JScript, keep in mind that the concepts are applicable to other scripting languages.

4.4.1 Command Line Arguments

The script begins by defining a set of command line arguments. These arguments are grouped into a global structure called testEnv to minimize namespace pollution. This structure also defines other variables such as boardname and cpuname for the purposes of logging this information globally.

```
testEnv = {
    // base path to .out, script, profile content etc
    basePath: WScript.Arguments(0),

    // Set test program (location of .out relative to basePath)
    testProgFile: WScript.Arguments(1),

    // Set test program filename, without the extension
    testProgFilename: WScript.Arguments(2),

    // Which profile config (.ini), could supply a list of profile configs to run
    profileConfig: WScript.Arguments(3),

    // do we wish to see CCStudio GUI during script (for test) or run in background?
    scriptingVisible: WScript.Arguments(4),

    // Paths to log file and profile results
    logFilePath: WScript.Arguments(5),
    profileResultsPath: WScript.Arguments(6),

    // breakpoint attributes required (fxn name)
    breakpointFxnName: WScript.Arguments(7),

    // ; delimited paths to source code for ATK
    atkSrcPath: WScript.Arguments(8),
```

```
    // CCStudio Install path
    ccsInstallDir: WScript.Arguments(9),

    // Sim files path (for port/pin connect files)
    simFilesPath: WScript.Arguments(10),

    // Which MCBSP? (eg dsk6416 uses MCBSP2, dsk6713 uses MCBSP1)
    mcbspNumber: WScript.Arguments(11),

    // Address of the DXR & DRR for the given MCBSP
    dxrDrrAddress: WScript.Arguments(12),

    boardName: "",
    cpuName: "",
    majorISA: "",
    minorISA: ""
};
```

The basePath variable defines the location of the script. Other relative paths begin with basePath.

The testProgFile variable defines the location of the .out file that is to be run, relative to the basePath. testProgFilename is the filename (without the .out extension) of the same executable file. This duplication is handy to have and helps eliminate some of the tricky string parsing that would be necessary otherwise.

The profileConfig variable defines the relative path to the profile configuration (.ini) file to be used for profiling. This file is created before running the script and should have the Collect Code Coverage and Exclusive Profile data option selected in order to gather the data necessary for the ATK.

The scriptingVisible variable is a flag to optionally visualize the CCStudio GUI during script execution (1 means visible, 0 means invisible). This is useful for debugging and demos. However, running CCStudio in the background is faster and should be the preferred option when doing “real” work.

The logFilePath and profileResultsPath variables are relative paths to the locations of the files produced by the run.

The breakpointFxnName variable specifies a location to put a CCStudio breakpoint. This is the location at which a CCStudio breakpoint would be set in the code so that when we run the code, we know where the program counter ends up. We set it at the start of the function thrRxSplitRun() so that it marks the beginning of processing of an audio frame. This function is called on each frame to perform pre-processing of input data. By running the code from one hit of this breakpoint to another, we can isolate the processing of one audio frame. See the Reference Frameworks documentation (Reference 3) for details on the functions in the framework.

The atkSrcPath variable defines the semi-colon delimited paths that point to the locations of the source code. This is used by the ATK’s code coverage component to show code coverage information on a line-by-line basis in the C sources. The ccsInstallDir variable needs to be specified to the script so that the script can find the ATK executables.

The simFilesPath, mcbspNumber, and dxrDrrAddress variables are necessary for running on the simulator. simFilesPath contains the necessary port and pin connect files for full McBSP peripheral simulation. mcbspNumber selects which McBSP is to be used for data streaming, and the dxrDrrAddress is the address corresponding to the DXR (data transmit register) and DRR (data receive register) of that particular McBSP, as accessed by the EDMA on the peripheral bus. See Reference 2 for details on how pin and port connect are used in RF3.

4.4.2 Logging Information and Starting CCStudio

The next portion of the script prepares scripting, logging, and CCStudio environments as follows:

```
var WshShell = WScript.CreateObject("WScript.Shell");
var ccs = WScript.CreateObject("CCS_Scripting_Com.CCS_Scripting");

// Open log file
ccs.ScriptTraceBegin(testEnv.basePath + testEnv.logFilePath + "\\log_file.txt");
ccs.ScriptTraceVerbose(VERBOSE_ALL);

// open whichever config is in CCSetup right now
ccs.CCSOpenNamed("*", "*", testEnv.scriptingVisible);

testEnv.boardName = ccs.TargetGetBoardName();
testEnv.cpuName = ccs.TargetGetCPUName();
testEnv.majorISA = ccs.TargetGetMajorISA();
testEnv.minorISA = ccs.TargetGetMinorISA();
ccs.ScriptTraceWrite("TEST: opened CCS connection: " + testEnv.boardName + " - " +
    testEnv.cpuName + " - " + testEnv.majorISA + " - " + testEnv.minorISA + "\n");

/* Output Command line arguments as sanity check */
PrintWScriptArgs();
```

After initializing the testEnv global structure, the script creates two objects, one for the Windows scripting shell and another for the CCStudio scripting engine. Using APIs offered by both, we can access all the functionality needed to automate profiling. The Windows scripting host also offers access to other Windows functionality through the export of other objects. (Reference 4).

CCStudio is launched by using the CCSOpenNamed() function. The call assumes the correct simulator is configured in CCStudio Setup and opens it up. It also determines whether the GUI will be hidden based on the scriptingVisible argument. A possible improvement to this script is to pass in the platform name to select a specific configuration to open in CCStudio. But we'll leave this as an exercise.

Scripting can log the success or failure of all automation calls to a file. ScriptTraceBegin() creates such a file. It is good practice to set the verbosity level to VERBOSE_ALL using ScriptTraceVerbose(). This gives the largest amount of information possible and can be useful in debugging the script. Furthermore, APIs such as TargetGetCPUName(), TargetGetMajorISA() and TargetGetMinorISA() are useful for adding identification information to a log file.

The PrintWScriptArgs() function call uses Windows Scripting Shell to print out the arguments to the script. This is solely done to facilitate script debugging.

4.4.3 Getting Ready for the Run

The next part of the script uses CCStudio Scripting APIs to prepare the environment for running the program as follows:

```
// reset CPU so that numbers are truly same on each run
ccs.TargetReset();
ccs.ScriptTraceWrite("TEST: reset\n");

// load the .out that we passed in via cmd line
ccs.ProgramLoad(testEnv.testProgFile);
ccs.ScriptTraceWrite("TEST: loaded\n");


// flip '\' to '/' in the basePath for passing paths to GEL expressions.
//   Because the usage of '\' requires a preceding escape character, we prefer
//   to bail and use '/' instead.
var basePathFwdSlash = testEnv.basePath.replace(/\\/g, "/");
testEnv.simFilesPath = testEnv.simFilesPath.replace(/\\/g, "/");

// do port connect for McBSP data
var tmp;
ccs.ScriptTraceWrite("TEST: Connecting Ports...\n");
tmp = basePathFwdSlash + testEnv.simFilesPath + "/input_data.txt";
ccs.TargetEvalExpression('GEL_PortConnect(' + testEnv.dxrDrrAddress + ', 1, 4, 1, "' + tmp +
'" )' );
tmp = basePathFwdSlash + testEnv.simFilesPath + "/output_data.txt";
ccs.TargetEvalExpression('GEL_PortConnect(' + testEnv.dxrDrrAddress + ', 1, 4, 2, "' + tmp +
'" )' );

// do pin connect for McBSP data
var pin;
tmp = basePathFwdSlash + testEnv.simFilesPath + "/CLKX" + testEnv.mcbspNumber + ".txt";
pin = "CLKX" + testEnv.mcbspNumber;
ccs.ScriptTraceWrite("TEST: Connecting Pins...\n");
ccs.TargetEvalExpression('GEL_PinConnect("' + pin + '", "' + tmp + '" )' );

tmp = basePathFwdSlash + testEnv.simFilesPath + "/CLKR" + testEnv.mcbspNumber + ".txt";
pin = "CLKR" + testEnv.mcbspNumber;
ccs.TargetEvalExpression('GEL_PinConnect("' + pin + '", "' + tmp + '" )' );

tmp = basePathFwdSlash + testEnv.simFilesPath + "/FSR" + testEnv.mcbspNumber + ".txt";
pin = "FSR" + testEnv.mcbspNumber;
ccs.TargetEvalExpression('GEL_PinConnect("' + pin + '", "' + tmp + '" )' );

tmp = basePathFwdSlash + testEnv.simFilesPath + "/FSX" + testEnv.mcbspNumber + ".txt";
pin = "FSX" + testEnv.mcbspNumber;
ccs.TargetEvalExpression('GEL_PinConnect("' + pin + '", "' + tmp + '" )' );
ccs.TargetEvalExpression('GEL_TextOut("Ports and pins successfully connected.\n");');

// set breakpoint at fxn we wish to begin profiling session from
var bpFxnAddress = ccs.SymbolGetAddress(testEnv.breakpointFxnName);
ccs.BreakpointSetAddress(bpFxnAddress);
```

The script resets the board, loads the executable, sets up pin and port connect using scripting APIs, and sets the breakpoint at the beginning of the code (thrRxSplitRun()) for processing an audio frame. The example uses TargetEvalExpression() to call GEL commands for port and pin connect. These synchronous calls are guaranteed to be completed by the time TargetEvalExpression() returns, as shown in Appendix A.

Note the technique of dynamically constructing the GEL calls executed with TargetEvalExpression(). If you are an experienced GEL user, you may recall that GEL files do not accept any form of command-line parameters. Scripting works around this limitation with the TargetEvalExpression() API. This allows you to eliminate hard-coded paths or filenames.

The syntax for TargetEvalExpression() could be confusing to read for some when used with GEL_PortConnect() and GEL_PinConnect(). When evaluated, they call GEL commands with the following syntax:

GEL_PortConnect(<some_address>, 1, 4, 1, “<some_file_path>”)
GEL_PinConnect("<some_pin_name>", "<some_file_path>")

Because double-quotes are used within the GEL commands themselves, we use single-quotes to delimit the different concatenated “chunks” that combine into the argument for TargetEvalExpression().

4.4.4 Run and Profile

Next, the script runs the program and profiles the code as follows:

```
// run to 1st breakpoint
ccs.TargetRun();
ccs.ScriptTraceWrite("TEST: ran to breakpoint\n");

// run to same breakpoint again to put the cache in steady state
ccs.TargetRun();
ccs.ScriptTraceWrite("TEST: ran to 2nd breakpoint\n");

// load the profile config ie Code Coverage & Exclusive counts selected
testEnv.profileConfig = testEnv.basePath + testEnv.profileConfig;
ccs.LoadProfileConfiguration(testEnv.profileConfig);

// enable profiling for this frame
ccs.EnableProfiling();

// run to same breakpoint again now collecting profiling data
ccs.TargetRun();
ccs.ScriptTraceWrite("TEST: ran to 3rd breakpoint\n");

// need to disable profiling in order to run Code Coverage viewer
ccs.DisableProfiling();

// remove any stale files from previous runs
WshShell.run("cmd /c del \"*.csv\"", 0, true);
testEnv.profileResultsPath = testEnv.basePath + testEnv.profileResultsPath;
WshShell.run("cmd /c del \"" + testEnv.profileResultsPath + "\\"
      + testEnv.testProgFilename + "_summary.xls\"", 0, true);

// Generate ATK results
ret_val = ATKtprof2xls(testEnv.ccsInstallDir,
             testEnv.testProgFile,
             testEnv.testProgFilename,
             tprofFilename,
             testEnv.profileResultsPath + "\\" + testEnv.testProgFilename + "_summary.xls",
             testEnv.atkSrcPath);

ccs.ScriptTraceWrite("TEST: ran ATK coverage & cov2xls. ret_val = " + ret_val + "\n");
```

In the script, TargetRun() is called three times:

- Once to run it to the thrRxSplitRun() breakpoint
- Another time to process the first frame which will populate the cache
- Once more for profiling

Before the last run, the script loads the profile configuration file via LoadProfileConfiguration(), and enables profiling.

After completion, the script disables profiling and calls the function ATKtprof2xls() to convert the .tprof information file produced by the ATK into an Excel spreadsheet, using command line executables provided in the CCStudio installation.

The ATK is used to profile RF3 because currently the CCStudio profiler is not designed for threaded applications. Since RF3 uses software interrupt (SWI) threads extensively, using the CCStudio profiler is not an option. The CCStudio profiler can be used only when profiling straight-line test code that runs in main(), with minimal context switches. In this case, Scripting provides an API named ExportProfileData() to use instead of ATKtprof2xls().

4.4.5 Finalization

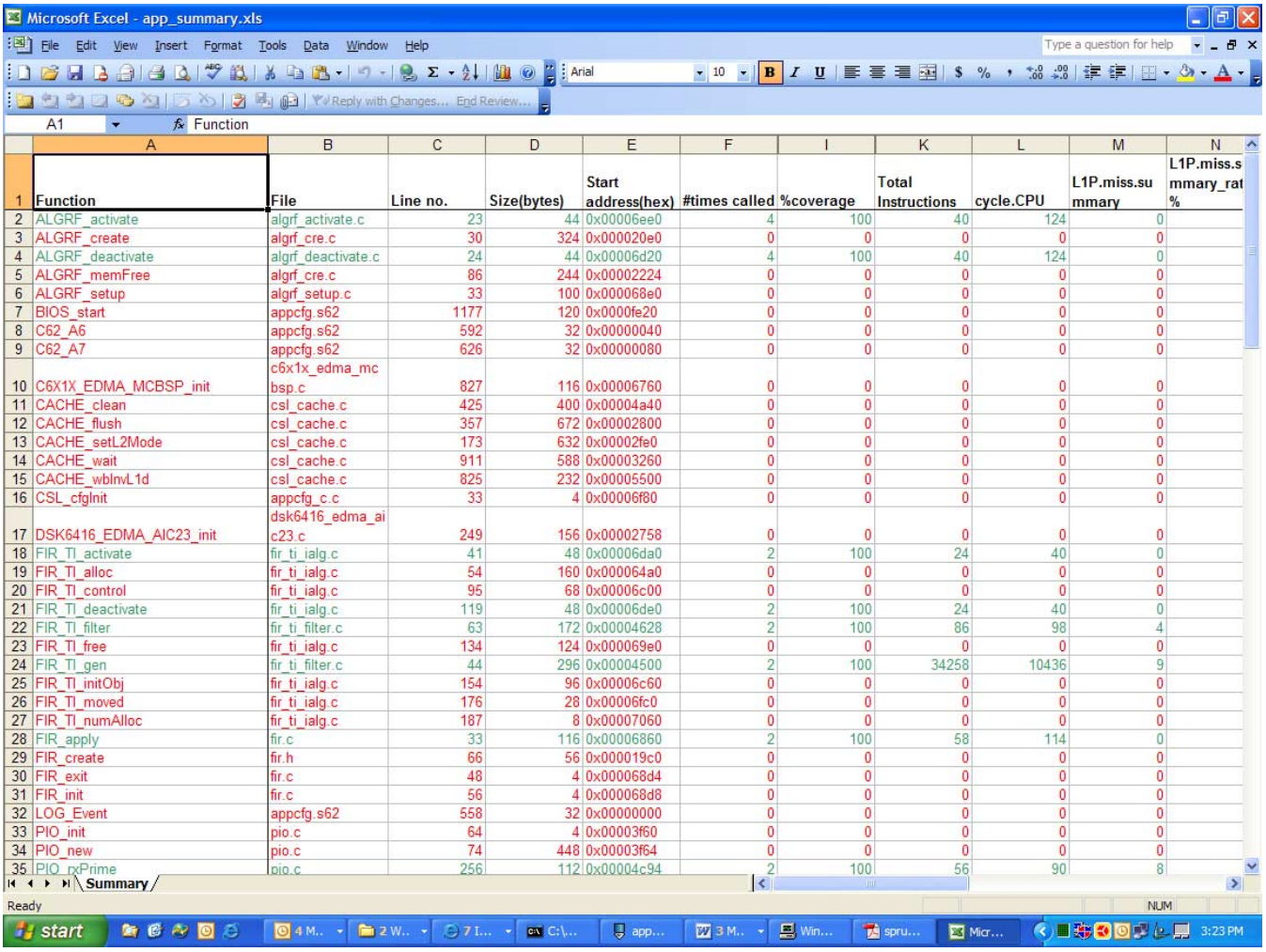
The script ends by clearing all breakpoints that were set as part of the procedure as follows:

```
ccs.BreakpointRemoveAll();  // Clear all breakpoints
ccs.ScriptTraceWrite("\nTEST: The End\n");
ccs.ScriptTraceEnd();    // Close log file
ccs.CCSClose(1);
```

This is a good habit, especially if you intend to load a different test program after this point, as existing breakpoints are unlikely to be useful for the next executable. Then the script closes the log file and the CCStudio instance. The latter is important because leaving the CCStudio executable running consumes valuable system resources.

4.5 Results

The profile data is produced in an Excel spreadsheet in the profile_results folder. A snapshot of the spreadsheet is shown in Figure 4.



Microsoft Excel - app_summary.xls

	A	B	C	D	E	F	I	K	L	M	N
1	Function	File	Line no.	Size(bytes)	Start address(hex)	#times called	%coverage	Total Instructions	cycle.CPU	L1P.miss.summary	L1P.miss.summary_rat %
2	ALGRF_activate	algrf_activate.c	23	44	0x00006ee0	4	100	40	124	0	
3	ALGRF_create	algrf_cre.c	30	324	0x000020e0	0	0	0	0	0	
4	ALGRF_deactivate	algrf_deactivate.c	24	44	0x00006d20	4	100	40	124	0	
5	ALGRF_memFree	algrf_cre.c	86	244	0x00002224	0	0	0	0	0	
6	ALGRF_setup	algrf_setup.c	33	100	0x000068e0	0	0	0	0	0	
7	BIOS_start	appcfg.s62	1177	120	0x0000fe20	0	0	0	0	0	
8	C62_A6	appcfg.s62	592	32	0x00000040	0	0	0	0	0	
9	C62_A7	appcfg.s62	626	32	0x00000080	0	0	0	0	0	
10	C6X1X_EDMA_MCBSP_init	c6x1x_edma_mcbsp.c	827	116	0x00006760	0	0	0	0	0	
11	CACHE_clean	csl_cache.c	425	400	0x00004a40	0	0	0	0	0	
12	CACHE_flush	csl_cache.c	357	672	0x00002800	0	0	0	0	0	
13	CACHE_setL2Mode	csl_cache.c	173	632	0x00002fe0	0	0	0	0	0	
14	CACHE_wait	csl_cache.c	911	588	0x00003260	0	0	0	0	0	
15	CACHE_wbInvL1d	csl_cache.c	825	232	0x00005500	0	0	0	0	0	
16	CSL_cfgInit	appcfg_c.c	33	4	0x00006f80	0	0	0	0	0	
17	DSK6416_EDMA_AIC23_init	dsk6416_edma_aic23.c	249	156	0x00002758	0	0	0	0	0	
18	FIR_TI_activate	fir_ti_ialg.c	41	48	0x00006da0	2	100	24	40	0	
19	FIR_TI_alloc	fir_ti_ialg.c	54	160	0x000064a0	0	0	0	0	0	
20	FIR_TI_control	fir_ti_ialg.c	95	68	0x00006c00	0	0	0	0	0	
21	FIR_TI_deactivate	fir_ti_ialg.c	119	48	0x00006de0	2	100	24	40	0	
22	FIR_TI_filter	fir_ti_filter.c	63	172	0x00004628	2	100	86	98	4	
23	FIR_TI_free	fir_ti_ialg.c	134	124	0x000069e0	0	0	0	0	0	
24	FIR_TI_gen	fir_ti_filter.c	44	296	0x00004500	2	100	34258	10436	9	
25	FIR_TI_initObj	fir_ti_ialg.c	154	96	0x00006c60	0	0	0	0	0	
26	FIR_TI_moved	fir_ti_ialg.c	176	28	0x00006fc0	0	0	0	0	0	
27	FIR_TI_numAlloc	fir_ti_ialg.c	187	8	0x00007060	0	0	0	0	0	
28	FIR_apply	fir.c	33	116	0x00006860	2	100	58	114	0	
29	FIR_create	fir.h	66	56	0x000019c0	0	0	0	0	0	
30	FIR_exit	fir.c	48	4	0x000068d4	0	0	0	0	0	
31	FIR_init	fir.c	56	4	0x000068d8	0	0	0	0	0	
32	LOG_Event	appcfg.s62	558	32	0x00000000	0	0	0	0	0	
33	PIO_init	pio.c	64	4	0x00003f60	0	0	0	0	0	
34	PIO_new	pio.c	74	448	0x00003f64	0	0	0	0	0	
35	PIO_rxPrime	pio.c	256	112	0x00004c94	2	100	56	90	8	

Figure 4. Snapshot of Profiling Results

The spreadsheet shows both code coverage data along with exclusive profiling results for cycle counts, cache hits/misses, and more.

Note that in addition to the Excel spreadsheet, additional intermediate files (.cov, .csv) are generated by the ATK. If you want to keep the data in these formats for data merging or post-processing, you can archive these files instead of the .xls file. See the ATK documentation for information on these formats.

5 Summary

The Code Composer Studio Scripting Utility was designed to provide the ability to automate unit- and system-level testing of an application with CCStudio. When used with a script that takes advantage of the basic features found in standard scripting languages, fully automated portable and configurable test benches (such as the example shown in Section 4) can be created to perform such critical actions such as benchmarking and regression testing of an application with CCStudio.

6 References

1. Code Composer Studio Scripting Guide on-line help and HTML-based documentation (available in the \bin\utilities\ccs_scripting\docs directory of your CCStudio installation).
2. *Simulating RF3 to Leverage Code Tuning Capabilities* (SPRAA73).
3. Reference Frameworks source code and documentation.
4. MSDN Library Windows Scripting website.

Appendix A. GEL Synchronicity Table

Below is a table that lists the synchronistic behavior of all the built-in GEL functions. If a GEL function is described as synchronous, then the Scripting API TargetEvalExpression() waits until the action associated with that GEL call has been completed. If a GEL function is not synchronous, then TargetEvalExpression() can potentially return before the action associated with the GEL call has been fully completed. In the latter case, you should look for an equivalent CCStudio Scripting API, otherwise results may be timing-dependent.

Table 1. GEL Synchronicity Table of Built-in GEL Functions

GEL Function	Synchronous?
GEL_AddInputFile	yes
GEL_AddOutputFile	yes
GEL_Animate	no
GEL_AsmStepInto	no
GEL_AsmStepOver	no
GEL_BreakPtAdd	yes
GEL_BreakPtDel	yes
GEL_BreakPtDisable	yes
GEL_BreakPtReset	yes
GEL_CancelTimer	yes
GEL_ClearProfileConfiguration	yes
GEL_CloseWindow	yes
GEL_DisableRealtime	no
GEL_EnableRealtime	no
GEL_Exit	no
GEL_Go	yes (only if location given)
GEL_Halt	no
GEL_HWBreakPtAdd	yes
GEL_HWBreakPtDel	yes
GEL_HWBreakPtDisable	yes
GEL_HWBreakPtReset	yes
GEL_IsInRealtimeMode	yes
GEL_Load	no
GEL_LoadGel	yes
GEL_MapAdd	yes
GEL_MapAddStr	yes
GEL_MapDelete	yes
GEL_MapOff	yes
GEL_MapOn	yes
GEL_MapReset	yes
GEL_MemoryFill	yes
GEL_MemoryLoad	yes
GEL_MemorySave	yes
GEL_OpenDisassemblyWindow	yes
GEL_OpenMemoryWindow	yes

GEL Function	Synchronous?
GEL_OpenWindow	yes
GEL_PatchAssembly	yes
GEL_PinConnect	yes
GEL_PinDisconnect	yes
GEL_PortConnect	yes
GEL_PortDisconnect	yes
GEL_ProbePtAdd	yes
GEL_ProbePtDel	yes
GEL_ProbePtDisable	yes
GEL_ProbePtReset	yes
GEL_ProjectBuild	no
GEL_ProjectBuildConfig	yes
GEL_ProjectClose	yes
GEL_ProjectCreateCopyConfig	yes
GEL_ProjectCreateDefaultConfig	yes
GEL_ProjectLoad	yes
GEL_ProjectRebuildAll	no
GEL_ProjectRebuildAllConfig	no
GEL_ProjectRemoveConfig	yes
GEL_ProjectSave	yes
GEL_ProjectSetActive	yes
GEL_ProjectSetActiveConfig	yes
GEL_RefreshWindows	no
GEL_RemoveDebugState	no
GEL_RemoveInputFile	yes
GEL_RemoveOutputFile	yes
GEL_Reset	yes
GEL_Restart	yes
GEL_RestoreDebugState	no
GEL_Run	no
GEL_RunF	no
GEL_SetSimMode	yes
GEL_SetTimer	yes
GEL_SharedMemHaltOnStepOff	yes
GEL_SharedMemHaltOnStepOn	yes
GEL_SharedMemHaltOnWriteOff	yes
GEL_SharedMemHaltOnWriteOn	yes
GEL_SrcDirAdd	yes
GEL_SrcDirRemoveAll	yes
GEL_SrcStepInto	no
GEL_SrcStepOver	no
GEL_StepInto	no
GEL_StepOut	no
GEL_StepOver	no
GEL_SymbolAdd	no
GEL_SymbolAddRel	no

GEL Function	Synchronous?
GEL_SymbolDisable	no
GEL_SymbolEnable	no
GEL_SymbolHideSection	no
GEL_SymbolLoad	no
GEL_SymbolLoadRel	no
GEL_SymbolRemove	no
GEL_SymbolShowSection	no
GEL_SyncHalt	no
GEL_SyncRun	no
GEL_SyncStepInto	no
GEL_SyncStepOut	no
GEL_SyncStepOver	no
GEL_System	no
GEL_TargetTextOut	yes
GEL_TextOut	yes
GEL_TransferToFile	yes
GEL_TransferToFileConfg	yes
GEL_UnloadAllSymbols	no
GEL_UnloadGel	yes
GEL_WatchAdd	yes
GEL_WatchDel	yes
GEL_WatchReset	yes
GEL_XMDef	yes
GEL_XMOff	yes
GEL_XMOn	yes

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265