Freescale Semiconductor
Application Note

Document Number: AN3595
Rev. 1, 4/2008

Basic TCL Scripting with the CodeWarrior™ for StarCore®-based DSP IDE

TCL is a simple programming language widely used for web applications, desktop GUI applications, testing and automation, databases, and embedded development. There are as many reasons to use TCL as there are developers, however the top reasons include that it is easy to learn and use and there are many helpful TCL resources widely available.

In this application note, you will learn to write simple TCL scripts to access the common features offered by the CodeWarrior™ IDE and Debugger. However, we recommend that you consult the CodeWarrior Help feature and one of the many available TCL books for complete information. You can obtain additional information about TCL at the website http://www.tcl.tk

Contents

1 Overview 2
1.1 Assumptions. 2
1.2 Requirements 2
1.3 Example Software 3
2 TCL Script Basics 4
3 Text and File Output 5
4 Looping and Data Structures 6
5 Running the Debugger 10
5.1 Building a CodeWarrior Project. 10
5.2 Starting the CodeWarrior Debugger 10
5.3 Running the CodeWarrior Debugger 10
5.4 Closing the CodeWarrior debugger 11
6 Modifying Source Files 11
7 Accessing Target Memory 12
7.1 Writing To The Target 13
7.2 Reading From The Target 13
8 Conclusion 15

© Freescale Semiconductor, Inc., 2008. All rights reserved.

freescale™ semiconductor

1 Overview

The purpose of this application note is to show how to operate some of the most common CodeWarrior™ Integrated Development Environment (IDE) features from a TCL script. In the context of this application note, the CodeWarrior IDE offers a command-line interface that supports the TCL scripting language to access some of its features. This can be very useful in situations for which several of the IDE features need to be used repetitively. For example, one such instance occurs when performing testing or profiling on a target supported by the CodeWarrior Debugger. The testing or profiling can use a TCL script to modify the source code iteratively, build the project, run the application on the target, and read some results to a log file.

The examples in this application note include using the following operations:

- Modifying a simple source file
- Building the CodeWarrior project
- Starting the debugger
- Running the application on a target.

In addition, the application note describes how to use loops to execute the operations repeatedly, as well as how to access memory on the target to write variables or read results.

1.1 Assumptions

The user is assumed to have the following minimum knowledge and experience:

- Elementary experience using a high-level programming language
- Basic experience using the CodeWarrior Integrated Development Environment for StarCore software

This application note comes with example software that uses the Freescale MSC8144ADS as the target. Knowledge of the MSC8144 functionality is not necessary.

1.2 Requirements

The software that accompanies this application note was developed on CodeWarrior for StarCore version 3.1 and this tool is required to perform the exercises described in this document. Furthermore, some of the exercises also require the use of an MSC8144 ADS board.

1.3 Example Software

A CodeWarrior for StarCore project is provided with this note to demonstrate the use of TCL scripts. This project example uses a simple application that consists of a number of channels, each with an associated buffer in memory. A simple C function called `ProcessBuff()` modifies the buffer array for each channel as indicated in the pseudo code listed in Example 1.

Example 1. `ProcessBuff()` Pseudo Code

```
ProcessBuff()
{
    For all CHANNELS
    {
        For all BUFFER_SIZE
        {
            Modify buffer array element
        }
    }
}
```

This C function is called from `main()` after initializing all channel buffers to zero. The main routine also sets up the MSC8144 Debug and Profiling Unit (DPU) which is used to capture profiling information about the function as shown in the pseudo code in Example 2.

Example 2. `main()` Pseudo Code

```
main()
{
    Initialize channel buffer and DPU
    Start DPU
    ProcessBuff()
    Read DPU
}
```

The application note shows you how to write a TCL script to run the example application using the following parameters:

- Number of Channels: 8, 16 and 32
- Buffer sizes: 1024, 2048, 4096, and 8192 bytes
- DPU profiling events: 0, 1, 2 and 3

The advantage of using a script is evident because there are 48 different combinations of parameters. Each combination requires the application to be built and run on the target. The advantage of automating this process is clear. The number of channels and the buffer size are compile-time parameters that are changed in a source file prior to building the application. The DPU profiling event is a run-time parameter that is written to a C variable in memory prior to executing the application. The profiling information is also read from memory after the application has executed and is written to a log file. The exercises in this application note build a TCL script that performs all these functions starting from the basic output of text strings to the full implementation

2 TCL Script Basics

Creating and running a TCL script is very easy with the CodeWarrior IDE. The CodeWarrior IDE Command Window is a command-line interface used to issue command lines to the IDE. For example, to start a debugging session use the `debug` command. You can use this command-line interface together with the TCL scripting engine to execute TCL scripts.

In the CodeWarror IDE tool suite, use the `View->Command Window` menu to bring up the window shown in Figure 1.

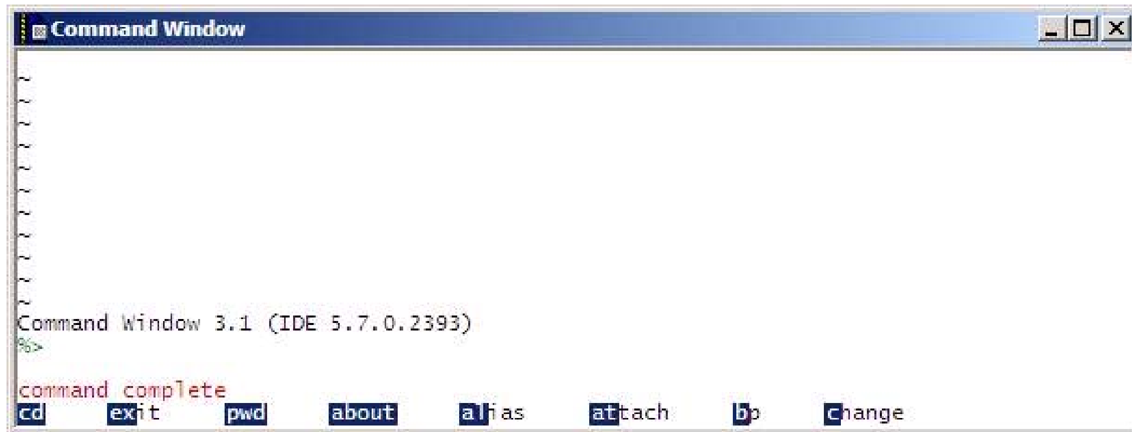

Figure 1. IDE Command Window

To issue a command, simply type the command in the Command Window, and press Enter or Return. The IDE executes the command that you entered. In addition, the Command Window displays the standard output and standard error streams of command-line activity.

Enter `help` in the Command Window to see a list of available commands and a brief explanation of each command. Enter `help` command to see a detailed explanation of the command.

Exercise 1

In the software provided with this application note, open the CodeWarrior project file for Core 0 (`MSC8144_core0.mcp`). Then open the Command Window using the `View->Command Window` menu, and type the `pwd` command. The Command Window displays the current working directory where the `MSC8144_core0.mcp` file is located.

To create a TCL script, simply open a text file and place the desired sequence of TCL and IDE commands in the file. Select a name for the file and add the extension `.tcl`. The extension `.tcl` is not required, but it is used by convention. Place the TCL script in a location where you can find it.

The TCL command `source` lets you run the sequence of commands that you place into the TCL script. For example, to run the TCL script called `DPU.tcl` type `source <path>/DPU.tcl` in the Command Window. The `<path>` is the directory path from the working directory to the TCL script. The working directory can be found by typing the `pwd` command in the Command Window. When a CodeWarrior project is opened, the

working directory automatically becomes the directory where the CodeWarrior project file (`.mcp`) is located.

Exercise 2

Create a TCL script, name it `DPU.tcl` and place it in the same directory as the CodeWarrior project for core 0 (`MSC8144_core0.mcp`) provided with this application note. In the script, enter the command `cls` which clears the Command Window screen. Comment lines in a TCL script are preceded by a pound (#) character. Use this symbol to add a comment describing what the `cls` command is doing. Save and close the TCL script, and then execute the script from the IDE Command Window using the `source` command. This should clear the screen from the output generated in Exercise 1.

Section 3, "Text and File Output" describes the commands for output of text to the standard output and to a file.

3 Text and File Output

The TCL `puts` command is used to print a string of text to a standard output device, which in our case is the IDE Command Window. The default behavior is to print a newline character ("return") after printing the text. If the string has more than one word, you must enclose the string in double quotes or braces ({ }).

Exercise 3

In the TCL script (`DPU.tcl`) created in the last exercise, add the command to output the string `Start DPU profiling` to the Command Window. Run the TCL script.

TCL allows you to store values in variables and use the variables in subsequent commands. The TCL command `set` is used to write and read variables. When `set` is called with two arguments, (for example, `set arg1 arg2`) it places the second argument (`arg2`) in the memory space referenced by the first argument (`arg1`). For example, the following command modifies the variable Y to hold the value 32:

```
set Y 32
```

For the purposes of this document, the first argument (`arg1`) is limited to a single word used to reference a text string in the second argument (`arg2`).

When a `$` appears in a command, TCL treats the letters and digits following it as a variable name, and substitutes the value of the variable in place of the name. Thus, because the argument in the `puts` command in the following code is preceded with `$`, TCL uses the value of the variable instead, and the result is that "Start DPU Profiling" is printed to the standard output as in Exercise 3:

```
set str1 "Start DPU Profiling"
puts $str1
```

TCL provides the `open` command to open a file and returns a token (`file ID`) to be used when accessing the file for input or output. The format of this command is:

```
open <filename> <access> <permission>
where
        <filename> is the name of the file to open.
        <access> is the file access mode:
             r......Open the file for reading. The file must already exist.
             r+...Open the file for reading and writing. The file must already exist.
             w.....Open the file for writing. Create the file if it doesn't exist, or
             set the length to zero if it does exist.
             w+..Open the file for reading and writing. Create the file if it doesn't
             exist, or set the length to zero if it does exist.
             a......Open the file for writing. The file must already exist. Set the
             current location to the end of the file.
             a+...Open the file for writing. The file does not exist, create it. Set
             the current location to the end of the file.
        <permission> is an integer to use to set the file access permissions. The default
        is rw-rw-rw- which is fine for the purposes of this note.
```

To close a file, use the `close` command, which closes the file referenced by the token (`file ID`) returned from a previously executed `open` command:

```
close fileID
```

Thus, in the code example below, we open a file called `DPU_results.txt` in the output subdirectory from the current working directory. Then, we write the string `Start DPU profiling` to the file.

```
set testlog [open output/DPU_results.txt w]
puts $testlog "Start DPU profiling"
```

Exercise 4

Continue with the TCL script (`DPU.tcl`) used in the last exercise and add the two commands in the example code above to output the string `Start DPU profiling` to the file `DPU_results.txt`. Do not forget to close the file with the `close` command and the file ID in testlog. (Hint: reference the fileID using `$testlog`). Verify that the output file was created and that it contains the proper text.

Much of the information presented in this section came from http://www.tcl.tk. Refer to this website for details on TCL variables and file IO.

Now that we have a way to send output to a standard device or to a file, we will describe how to loop through a list of variables that we will set up to index through the various parameters listed for the application described in Section 1.3, "Example Software."

4 Looping and Data Structures

In TCL, the basic data structure is called a `list`. A `list`, as the name implies, is simply an ordered collection of numbers, words, strings, and so on. Lists can be created in several ways but for the purposes of this note we will use the `list` command with the following syntax:

```
set <list_name> [list "item 1" "item 2" "item 3"]
```

For example, the code in Example 3 creates a list for the number of channels in the application example described in Section 1.3, "Example Software". In Example 3, we declare (using the `set` command) a list (using the list command) called `channel_list` with three items, namely: 8, 16 and 32.

Example 3. List Code

```
#Set values to use for test_variable1
set channel_list [list 8 16 32 ]
```

The items in a list can be iterated through using the TCL command as shown in Example 4.

Example 4. foreach Code

```
foreach varname list {
body
}
```

The `foreach` command executes the `body` code one time for each item in `list`. On each pass, the variable `varname` contains the value of the next list item.

The code shown in Example 5 creates a loop that iterates through the number of channels in the list created in the code shown in Example 3, and prints the text `#define CHANNELS` followed by the channel number to the CodeWarrior Command Window.

Example 5. Loop Code

```
foreach channel_num $channel_list {
    set str1 "#define"
    set str2 "CHANNELS"
    set str3 $channel_num
    puts "$str1 $str2 $str3"
}
```

Exercise 5

Continue with the TCL script (`DPU.tcl`) used in the last exercise and add the two code snippets above to output the string #define CHANNELS followed by the channel number to standard output. The output seen on the CodeWarrior Command Window should be similar to the shown in Figure 2. Now print the same text in the output log file opened in Exercise 4.

```
~
~
~
~
Start DPU profiling
#define CHANNELS 8
#define CHANNELS 16
#define CHANNELS 32
%>

command complete
cd    exit    pwd    about    alias    attach    bp    change
```

Figure 2. Example Loop Definition

As with other programming languages, TCL allows the nesting of loops.

Exercise 6

Continue with the TCL script (`DPU.tcl`) used in the last exercise and create another list for the buffer sizes (1024, 2048, 4096, and 8192 bytes) in the application example described in Section 1.3. Nest a `foreach` command within the loop created in the previous exercise and output the string `#define BUFFER_SIZE` followed by the buffer size number to standard output. The output seen on the CodeWarrior Command Window should be similar to the one shown in Figure 3. Now print the same text in the output log file opened in Exercise 4.

```
Command Window
~
~
Start DPU profiling
#define CHANNELS 8
#define BUFFER_SIZE 1024
#define BUFFER_SIZE 2048
#define BUFFER_SIZE 4096
#define BUFFER_SIZE 8192
#define CHANNELS 16
#define BUFFER_SIZE 1024
#define BUFFER_SIZE 2048
#define BUFFER_SIZE 4096
#define BUFFER_SIZE 8192
#define CHANNELS 32
#define BUFFER_SIZE 1024
#define BUFFER_SIZE 2048
#define BUFFER_SIZE 4096
#define BUFFER_SIZE 8192
%>

command complete
cd     exit     pwd     about     alias     attach     bp     change
```

Figure 3. Nested Loop Definition

Exercise 7

Continue with the TCL script (`DPU.tcl`) used in the last exercise and create one more list for the DPU profiling events (0, 1, 2, and 3) in the application example described in section 1.3. Nest another foreach command within the loop created in the previous exercise and output the string DPU Event followed by the profiling event number to standard output. The output seen on the CodeWarrior Command Window should be similar to the one shown in Figure 4. Now print the same text in the output log file opened in Exercise 4.

```
Command Window
Start DPU profiling
#define CHANNELS 8
#define BUFFER_SIZE 1024
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define BUFFER_SIZE 2048
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define BUFFER_SIZE 4096
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define BUFFER_SIZE 8192
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define CHANNELS 16
#define BUFFER_SIZE 1024
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define BUFFER_SIZE 2048
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define BUFFER_SIZE 4096
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define BUFFER_SIZE 8192
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
#define CHANNELS 32
#define BUFFER_SIZE 1024
DPU Event 0
DPU Event 1
DPU Event 2
DPU Event 3
%>

page 1 of 2 (press space for next, ESC to cancel; see also "
cd      exit     pwd     about     alias     attach     bp
```

Figure 4. Multiple List Definition

Section 5, "Running the Debugger" describes how to use the TCL commands to build an application and then run it in the CodeWarrior debugger.

5 Running the Debugger

This section presents the CodeWarrior commands used in a TCL script to build an application, load and run it on a target with the CodeWarrior debugger, and then close the CodeWarrior debugger.

5.1 Building a CodeWarrior Project

The `make` command builds a project. It uses the following syntax:

```
make [-e|-v|-n] [<project>.mcp]
where
        -e|-v|-n controls the types of messages shown. If unspecified, errors and
        warnings are shown.
             -e only errors are shown
             -v all messages are shown (verbose)
             -n no messages be shown
        <project>.mcp is the project to be built. If unspecified, the default (open)
        project is built.
```

5.2 Starting the CodeWarrior Debugger

The `debug` command starts the CodeWarrior debugger. It uses the following syntax:

```
debug [<project>.mcp]
where
        <project>.mcp is the project to be built. If unspecified, the default (open)
        project is built.
```

5.3 Running the CodeWarrior Debugger

The `go` command runs an application loaded in the CodeWarrior debugger. It uses the following syntax:

```
go [all]
where
        all to run all threads
```

The `wait` command is generally used in conjunction with the `go` command. It uses the following syntax

```
wait [milliseconds]
where
        milliseconds is the time to wait in milliseconds. If unspecified, wait until the
        user hits the SPACE bar.
```

The wait command is useful when you want to cause the subsequent command to execute after some delay following the preceding command. For example, in the code in Example 6, the `wait` command causes `command2` to execute 8000 ms after `command1`.

Example 6. `wait` Code

```
command1
wait 8000
command2
```

5.4 Closing the CodeWarrior debugger

The kill command closes the CodeWarrior debugging session. It uses the following syntax:

```
kill [all]
where
        all to close all threads
```

Exercise 8

Continuing with the TCL script (`DPU.tcl`) used in the last exercise, add within the inner-most `foreach` command loop created in the previous exercise the commands to build the application (only display errors), start the debugger, run the application and then kill the debugging session. Wait 5 seconds (5000 ms) between running the application and closing the debugger.

The script created in Exercise 8 runs the example application provided with this document in the debugger one time for each of the parameters described in Section 1.3, “Example Software” (number of channels, buffers sizes, and DPU profiling events). The Command Window displays the text added in Section 4, “Looping and Data Structures” indicating the application parameters. Similarly, the output text file created by the script shows these same text.

In addition, the application outputs text to a standard I/O window each time it runs indicating the number of channels, buffers sizes, and DPU profiling events used in that run. However, note that this text output indicates that these parameters are not changing. This is the case because these parameter are not being changed in the application source files or memory. Thus, the commands in the script always execute the application with the same parameters. Section 6, “Modifying Source Files” shows how to modify source files in the application.

6 Modifying Source Files

To have a TCL script modify the source files in an application, we use the same commands presented in Section 3, “Text and File Output” to output text strings to a file, namely `open`, `set`, `put`, and `close`. However, we also need to indicate to the CodeWarrior IDE that the source file is changed so that the `make` command builds the application using the new source file. To do this, use the `project` command that allows you to open a project, set the default project target, or add or remove files in a project. The following syntax is used to add and remove files from a CodeWarrior project:

```
project add|rm <file>… [-g[roup] <group>]
where
        add to add a file(s) to the project
        rm to remove a file(s) form the project
        file indicates the file name(s) to be added or removed
        group indicates the group in the CodeWarrior project to add or remove the files
        in question
```

Using the `project` command without any arguments returns the name of the default project and target.

The code in Example 7 removes the C source header file `buffersize.h` in the source directory from the default project, then opens that file, writes the new information to the file, closes the source file, and adds the modified header file to the project.

Example 7. Removing the C Source Header

```
set str4 "#define"
set str5 "BUFFER_SIZE"
set str6 $buffer_size

project rm ../source/buffersize.h
set outfile [open ../source/buffersize.h w]
puts $outfile "$str4 $str5 $str6"
close $outfile
project add ../source/buffersize.h -g Include
```

In Example 7, the value of variable `str6` is set to a variable called `buffer_size`. If `buffer_size` is the variable name used in the context of the `foreach` command, then the code in Example 7 can be used to modify a source file as it iterates through each of the items in TCL list.

Exercise 9

Add the code in Example 7 to the TCL script (`DPU.tcl`) used in the last exercise at the beginning of the `foreach` command loop that iterates through the buffer size parameter to modify the C source header file `buffersize.h`. The file is written with the C statement `#define CHANNELS` command followed by the channel number for that loop iteration. Run the TCL script and notice that for iterations in which the buffer size is changed by the TCL script that the `STDIO` text output from the application reflects this change.

Exercise 10

Now add commands similar to those added in Exercise 9 to the TCL script (`DPU.tcl`) within the `foreach` command loop that iterates through the number of channels parameter to modify the C source header file `channels.h`. Run the TCL script and notice that both the buffer size and the number of channels change in the `STDIO` text output from the application.

Now the two outer `foreach` loops in the `DPU.tcl` script change the buffer size and the number of channels of the example application at build time. In Section 7, “Accessing Target Memory,” we present how to change a variable in memory during run-time. We will use this method to change the DPU profiling events as we iterate through the list in the inner-most foreach loop of the script. We will also describe how to read memory to access the profiling results for each execution of the application. These results are written to the output log file for future parsing and analysis.

7 Accessing Target Memory

The last step we will discus is the reading and writing of variables in the target's memory. As mentioned earlier, this allows us to write the DPU profile parameters for the application listed in Section 1.3, “Example Software” and to read the profiling results from memory.

7.1 Writing To The Target

There are several CodeWarrior commands available to access registers, memory, variables or C expressions in the target. Some of these include `var`, `change`, `mem`, and `display`. For this discussion we will use the var command with the following syntax:

```
var <var> [%<conv>] =<value>
where
    [none] If no options are provided, then all variables pertinent to the current
    scope are printed.
    <var> Symbolic name of the variable to print. Can be a C expression as well.
    %<conv> Specifies the type of the data. Possible values for <conv> are given
    below. If unspecified, %x is used.
        %x Hexadecimal.
        %d Signed decimal.
        %u Unsigned decimal.
        %f Floating point.
         %[E<n>]F Fixed or Fractional.
        %s Ascii.
```

For example, the following code writes the C variable `gui32ProfileIndex` with the value of the TCL variable `$DPU_event`.

```
var gui32ProfileIndex = $DPU_event
```

Exercise 11

Add the `var` command to the TCL script (`DPU.tcl`) within the inner-most `foreach` command loop that iterates through DPU profiling events between the `debug` command and the `go` command. Write the variable `gui32ProfileIndex` with the value of the current item in the DPU profiling event list. Run the TCL script and notice that now the buffer size, the number of channels, and the profiling event change in the `STDIO` text output from the application.

7.2 Reading From The Target

As with the writing to the target, there are several CodeWarrior commands available to read registers, memory, variables, or C. The `evaluate` command displays the value of a variable or expression using the following syntax:

```
evaluate <var>
where
        [none] If no arguments are provided, then all variables pertinent to the current
        scope are printed.
        <var> is the symbolic name of the variable to print.
```

For example, the first command in the following code prints the value of the variable `gui32ProfileIndex` to the Command Window. The second command prints the variable `gui32ProfileIndex` to the output file referenced by `$testlog`.

```
evaluate gui32ProfileIndex
puts $testlog [evaluate gui32ProfileIndex]
```

Exercise 12

From the Command Window start the CodeWarrior debugger for the project provided with this note and run the application (see Section 5, “Running the Debugger”). Then, from the Command Window use the `evaluate` command to view all the value and type of the variables pertinent to the current scope.

Now use the `evaluate` command to view the value of the `gui32ProfileIndex` variable. Is this what you expect?

Use the `evaluate` command once again to view the value of the `gauliProfileResults` variable. Is this what you expect?

The `gui32ProfileIndex` variable is of type `unsigned long` and thus, the value returned is a scalar; however, the `gauliProfileResults` variable is an array, so the value returned is an address. For the example application, the DPU profiling results that we wish to access are stored in the `gauliProfileResults[4][3]` variable. The first dimension of the array `[4]` corresponds to each of the four cores on the MSC8144 target. The second dimension `[3]` corresponds to the 3 values typically provided from the DPU profiling event.

To access the values in the `gauliProfileResults[4][3]` array, we use the `evaluate` command in conjunction with the `mem` command. The `mem` command reads or writes one or more contiguous blocks of bytes in memory. The syntax we use for the `mem` command to read memory is as follows:

```
mem <addr> [<count>] [<width>] [%<conv>]
where
    <addr> The target address in hex.
    <count> The number of memory cells to read
    <width> The size in bits of each cell read (8, 16, 32, or 64)
    %<conv> Specifies the type of the data. Possible values for <conv> are given
    below. If unspecified, %x is used.
        %x Hexadecimal.
        %d Signed decimal.
        %u Unsigned decimal.
        %f Floating point.
         %[E<n>]F Fixed or Fractional.
        %s Ascii.
```

The code in Example 8 reads the `gauliProfileResults[4][3]` array with the DPU profiling results from the example application.

Example 8. Code to Read the Results Array

```
#Access profiling results for Core 0 - gauliProfileResults[0][3]
set resultaddress [evaluate gauliProfileResults]
puts $testlog [mem $resultaddress 3 32bit %d]

#Access profiling results for Core 1 - gauliProfileResults[1][3]
set resultaddress [evaluate gauliProfileResults+1]
puts $testlog [mem $resultaddress 3 32bit %d]

#Access profiling results for Core 2 - gauliProfileResults[2][3]
set resultaddress [evaluate gauliProfileResults+2]
puts $testlog [mem $resultaddress 3 32bit %d]

#Access profiling results for Core 3 - gauliProfileResults[3][3]
set resultaddress [evaluate gauliProfileResults+3]
puts $testlog [mem $resultaddress 3 32bit %d]
```

Exercise 13

Add the code in Example 8 to the TCL script (`DPU.tcl`) to write the DPU profiling results to the output text file. Paste the commands after the application has run (but before killing the debugging session). Run the TCL script and sit back and enjoy.

8 Conclusion

This application note teaches you how to write a TCL script that performs the following tasks in the accompanying example application software:

1. Modify a simple source file in the CodeWarrior project
2. Build the CodeWarrior project
3. Start a CodeWarrior debugging session
4. Write a variable in memory on the target
5. Run the application in the CodeWarrior debugger
6. Read resulting variables from the target's memory
7. Save results to an output log file

The examples demonstrate the usefulness of using TCL scripts in situations in which several tasks must be repeated several times, such as profiling an application. TCL is simple enough to become useful very quickly in these situations and the CodeWarrior IDE provides a simple interface to access its features.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3595
Rev. 1
4/2008

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™, the Freescale logo, StarCore, and CodeWarrior are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc., 2008. All rights reserved.

freescale™
semiconductor