Application Report
SPRAA67 – November 2004

DSP/BIOS Tconf Language Coding Standards

Steve Connell *Texas Instruments, Santa Barbara*

ABSTRACT

This document describes the Tconf language coding standards. This standard describes general lexical conventions. The benefits of these coding standards include preventing namespace collisions, simplifying the use of modules, and improving productivity.

Third-party software vendors are encouraged to follow these coding standards. These coding standards are suggested, but not mandatory, for third parties seeking eXpressDSP compliance certification.

Contents

Overview

DSP/BIOS uses the Tconf language for configuration. Tconf is based on JavaScript. Information about the Tconf language is provided in the *DSP/BIOS Textual Configuration (Tconf) User's Guide* (SPRU007). This document recommends coding standards for using Tconf.

This standard describes design conventions for general lexical conventions for use in Tconf configuration scripting. These coding standards have been used and improved upon by Texas Instruments, Santa Barbara since 1990. Third-party software vendors are encouraged to follow these standards. The benefits of these coding standards include:

- Preventing namespace collisions between separate modules.
- Simplifying the use of modules. If the standard is used, programmers know where to find symbol definitions and where to create and use instance objects.
- Simplifying code audits and improving productivity. Familiarity with the structure and conventions makes reviewing code created by others easier.
- Enabling tool support. If standards are followed, simple tools can be created to perform routine tasks.

Modules

A software system consists of a hierarchy of modules. Each module encapsulates a collection of related constants, variables, and functions. Although there are many pre-defined DSP/BIOS and Tconf modules available to you, you may find it necessary to create your own module. While similar to modules in C, a Tconf module definition should contain both the interface and the implementation of its functions in the same file.

A few terminology definitions are helpful in understanding modular design:

- **Module.** The smallest logical unit of software. A module should contain all of its functions (as well as their implementations) and variables in the same file.
- **Client application.** A program that calls interface functions to perform application work. This will be either your *.tcf file or one of your *.tci files.
- **Interface.** A collection of related constants, types, variables, and functions.
- **Implementation.** The source code for the functions in the module.

Module Lexical Conventions

Each module should be defined in a file that is named after the module name. For example, if your module is called “myModule,” then it should be defined in a file called “myModule.tci.” All identifiers and functions defined within this module should be pre-fixed with the name of the module. For example, the contents of myModule.tci would look something like this:

```
myModule.myInteger = 1;
myModule.myString = “foo”;

myModule.myFunction = function (argument1, argument2)
{
   ...
}
```

Variables defined within the module must be prefixed with the name of the module in order to prevent namespace collisions. Module variables do not need to be created using the var keyword.

Tconf (and JavaScript) allow you to declare for local and global variables. To make a variable local, you must use the var keyword in the declaration, and the variable must be declared in a function body. For example, in the following code, only myModule.var_c is a local variable.

```
/* myModule.tci */

var myModule.var_a = 0;
myModule.var_b = 0;

myModule.myFunction = function (argument1, argument2)
{
   var myModule.var_c = 0;
   myModule.var_d = 0;

}
```

Tconf variables defined within a module, and outside a function body, are always global variables. If a global variable foo is defined in myModule.tci, and is later defined in a different file, the myModule.tci instance of foo would be overwritten!

Local variables, such as myModule.var_c in the previous example, may be named without a module prefix.

Module Design Conventions

Tconf module files should contain the following contents:

1. **Banner.** Every module should contain a banner comment that contains a description of the module.

2. **Module object.** A module should begin by creating that module's object, and this object should be named after the module file name. The module object should always be created first and should not be placed in the variable list alphabetically. For example, the module object for myModule.tci should be created before anything else is done in the file:

   ```
   var myModule = {};
   ```

   If the module has any internal functions, the "internal" object used to contain these functions should be declared immediately after the module's object. See item 6 for information about internal functions.

3. **Variable declarations.** All module variables should next be defined and listed in alphabetical order, following the creation of the module object. (Note that local variables defined within functions should not be included in this list.)

4. **Function declarations.** All functions should next be defined in alphabetical order, and these definitions should start after the last module variable definition.

5. **Init function.** All modules should have an "init" function, which must be called explicitly at the end of the module's interface. This is required even if the module being defined does not need an init function. In this case, an empty init function should be created. For example, the init function in myModule.tci may look like:

```
myModule.init = function()
{
}
```

At the end of the module file, the init function should be called like this:

```
myModule.init();
```

6. **Internal functions.** If a module defines a function that is internal to the module, then that function should be defined through the "internal" object. If you need to create an internal function, you should first create the internal object, which should be done immediately after the creation of the module object:

```
var myModule.internal = {};
```

Internal functions should be defined using the internal object:

```
myModule.internal.myInternalFxn = function()
{
}
```

Functions that are defined as internal are usually created by the module's designer. These functions are normally used for special operations. An internal function should not be used outside of the module that it is created in.

Here is an example of what myModule.tci might look like once complete:

```
/*
 *  myModule.tci
 */

/* define the myModule object */
var myModule = {};

/* define myModule's variables in alphabetical order */
myModule.myInteger = 1;
myModule.myString = "hello";

/* define myModule's functions in alphabetical order */
myModule.foo = function()
{
}

myModule.init = function()
{
}
```

```
myModule.xyz = function()
{
    /*
     *  These variables do not need to be declared in the
     *  alphabetical list above because they are local to this function.
     */
    var x;
    var y;
    var z;
    x = y = z = 2;
}

/* explicitly call the init function */
myModule.init();
```

Module Source Files

Here is an example of a *.tcf and two *.tci files taken from the DSP/BIOS hostio example. Follow the layout of these files, which is explained in the next section, when creating your Tconf scripts.

TCF file: hostio.tcf. The lines that import the TCI files are highlighted.

```
/*
 *  Copyright 2004 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *
 *  @(#) DSP/BIOS_Examples 5,0,0 05-03-2004 (biosEx-a16)
 */
/*
 *  ======== hostio.tcf ========
 *  Configuration script used to generate the example's configuration files
 */

/*
 *  Import the dsk6713 settings that are common to most examples.
 */
utils.importFile("dsk6713_common.tci");

/*
 *  Import the hostio configuration settings that apply to all platforms.
 */
utils.importFile("hostio.tci");

if (config.hasReportedError == false) {
    prog.gen();
}
```

TCI file: dsk6713_common.tci

```
/*
 *  Copyright 2004 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *
 *  @(#) DSP/BIOS_Examples 5,0,0 05-03-2004 (biosEx-a16)
 */
/*
 *  ======== dsk6713_common.tci ========
 *  Custom settings that apply only to the dsk6713 platform.
 */

/*
 * Setup platform-specific memory map, CLK rate, etc.
 */
var mem_ext = [];
mem_ext[0] = {
    name: "SDRAM",
    base: 0x80000000,
    len:  0x00800000,
    space: "code/data"
};

var device_regs = new Object();
device_regs = {
    l2Mode: "SRAM"
};

var params = new Object();
params = {
    clockRate: 225.0000,
    catalogName: "ti.catalog.c6000",
    deviceName: "6713",
    regs: device_regs,
    mem: mem_ext
};

/*
 * Customize generic platform with parameters specified above.
 */
utils.loadPlatform("ti.platforms.generic", params);

/*
 * Enable common BIOS features used by all examples
 */
bios.enableRealTimeAnalysis(prog);
bios.enableMemoryHeaps(prog);
bios.enableRtdx(prog);
bios.enableTskManager(prog);

/*
 *  Define an initialization function to be called before hitting main().
 */
bios.GBL.CALLUSERINITFXN = true;
bios.GBL.USERINITFXN = prog.extern("GBL_setPLLto225MHz");
```

```
/*
 *  Turn on heaps in SDRAM and define label SEG0 for heap usage if this
 *  program has useMemSettings defined as true.
 */
bios.SDRAM.createHeap = true;
bios.SDRAM.heapSize = 0x8000;
bios.SDRAM.enableHeapLabel = true;
bios.SDRAM["heapLabel"] = prog.decl("SEG0");
```

TCI file: hostio.tci

```
/*
 *  Copyright 2004 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *
 *  @(#) DSP/BIOS_Examples 5,0,0 05-03-2004 (biosEx-a16)
 */
/*
 *  ======== hostio.tci ========
 *  Tconf include file imported by hostio.tcf which sets up global
 *  platform independent BIOS objects, properties, and parameters.
 *
 */

/*
 * Create and initialize two HST objects one for input, and one for output
 */
var input;
input           = bios.HST.create("input");
input.frameSize = 64;
input.numFrames = 2;
input.bufAlign  = 4;
input.mode      = "input";
input.notifyFxn = prog.decl("inputReady");

var output;
output           = bios.HST.create("output");
output.frameSize = 64;
output.numFrames = 2;
output.bufAlign  = 4;
output.mode      = "output";
output.notifyFxn = prog.decl("outputReady");

/*
 * Enable all DSP/BIOS components
 */
var copySwi;
copySwi         = bios.SWI.create("copySwi");
copySwi["fxn"]  = prog.decl("copy");
copySwi.mailbox = 3;
copySwi.arg0    = prog.decl("input");
copySwi.arg1    = prog.decl("output");
```

```
/*
 * Create and initialize a LOG object
 */
var trace;
trace = bios.LOG.create("trace");

/*
 * Set the buffer length of LOG_system buffer
 */
bios.LOG_system.bufLen = 512;
```

General Source File Layout

The following subsections discuss how Tconf code should be grouped and organized.

Grouping Code

Tconf code should be grouped into two different categories – platform-independent code and platform-dependent code.

- **Platform-independent files** (stored as *.tci files) should contain code that is (or can be) common to more than one ISA.
  - These files may be imported by a *.tcf script or an appropriate platform-dependent script.
  - They should be named in the same manner your C files are named. If your C files are named based on functionality, then you should name your Tconf files based on their functionality as well. For example, a file that sets up DSP/BIOS PIP objects may be called “myPipes.tci".
  - They should be shared by all ISAs that need them and stored in a common directory to avoid having duplicate files in different directories.
- **Platform-dependent files** contain code that is hardware-specific for the ISA you are working with. Some *.tci files often fall into this category. The *.tcf script is typically also platform-dependent, since it usually imports a platform-dependent file. (However, it is possible to designed a platform-independent *.tcf file using command-line arguments.)

Note: The *.tcf file is used to generate configuration files, and should import both platform-independent and platform-dependent *.tci files. All configuration file generation is accomplished through a *.tcf file; configuration files cannot be generated by a *.tci file alone.

Organizing Code

The code of your *.tcf script should be maintained using the following organizational groups. If the amount of code per group is large, it should be organized into Tconf include files (*.tci) and categorized as either platform-independent or platform-dependent (see the “Grouping Code” section).

1. Define all global variables. If these variables are organized into a *.tci file, that file should be imported using the utils.importFile() function.

2. Load the hardware platform as soon as possible. To load the hardware platform, use the utils.loadPlatform() function, which can be called with either of the following syntaxes:

   – **Recommended.** This syntax is usually called by a platform-dependent *.tci file, which first defines the params object immediately before calling this function. If this version of the function is used, the *.tcf script should import the *.tci file that calls utils.loadPlatform() as early as possible. See the hostio.tcf and dsk6713_common.tci files shown in the "Module Source Files" section on page 4.

   ```
   utils.loadPlatform(“ti.platforms.generic”, params);
   ```

   – **Older syntax.** This syntax is usually called directly from the *.tcf script. If this is the case, this function call should be made as early as possible. This form can also be called from within a *.tci file, and if this is the case, this *.tci file should be imported as early as possible within the *.tcf script.

   ```
   utils.loadPlatform(“Dsk5510”);
   ```

   See the *DSP/BIOS Textual Configuration (Tconf) User's Guide* (SPRU007) for information about how platform files referenced by utils.loadPlatform() are located.

3. Set up all platform-dependent properties, DSP/BIOS objects, variables, and parameters. If these things are organized into a *.tci file, then it should be imported next.

4. Set up all platform-independent properties, DSP/BIOS objects, variables, and parameters. If these things are organized into a *.tci file, then it should be imported next.

5. Call prog.gen() at the end of the file. It is recommended that you perform an error check before calling the prog.gen() function as follows:

   ```
   if (config.hasReportedError == false) {
      prog.gen();
   }
   else {
      throw new Error("Error in config script: Generated files have not been
   created.");
   }
   ```

   Although the prog.gen() function has no parameters, you are allowed to specify a directory as a parameter to this function. If you pass a directory, the configuration files are placed in that directory. If you specify no arguments, the configuration files are placed in the current directory. Note that if you specify a different directory location, your project or makefile must reference that location to find files needed for the build.

General Lexical Conventions

The following lexical conventions apply to all Tconf source files. In general, lexical conventions for Tconf are the same as the C language lexical conventions. See the guidelines defined in *C Language Coding Standards for DSP/BIOS and XDAIS* (SPRA788) for more details.

Identifiers

Variables are not explicitly typed in JavaScript because dynamic typing is used. This means that a variable's type is determined at run-time by the type of value being assigned to it. For example:

```
var myInteger = 10; // integer variable
var myString = "programming is fun"; // string variable
```

- Variable and function names consist of an appended series of whole or abbreviated words. The first word is entirely in lowercase. Subsequent appended words usually begin with an uppercase letter. Underscores are not used between words; they only follow the module prefix. For example:

```
var  countNumberFrames;
```

- Language keywords and ';' are always followed by one space.

```
for (i = 0; i < 10; i++)
   argument1 += argument2;
}
```

See the "Declarations" section on page 11 for more information on identifiers.

Expressions

- No space exists between unary operators or between the primary operators and their operand.
  - Unary operator examples:

    `-17`
  - Primary operator examples:

    `a[i]`

    `s.member`

    `f(x, y)`
- There is exactly one space between other operators and their operands. For example:

```
x = f(y) * (z + 2);
```

- Parentheses are used for clarity, especially in expressions involving several precedence levels.
- Unless explicitly defined by the JavaScript language, evaluation order is not relied upon.

Statements

- All subordinate statements are enclosed in braces and indented four (4) spaces.
- There is exactly one statement per line.
- Continuation lines are indented eight (8) spaces

Declarations

- Each variable or member is declared on a separate line:

```
var x;
var y;
```

Although you can declare variables as shown above in JavaScript, it is not entirely necessary because the language is dynamically typed. This means that the type of the variable is determined at run-time, and more than one type can be stored in the same variable. For example, the following code is legal:

```
var myInteger = 100; /* myInteger is determined to store an int at run time */

var myInteger = “hello world”;
/* myInteger has been determined dynamically to be of type string and will store
the string value “hello world”, overwriting the previous integer value of 100. */
```

- It is recommended that variables be declared using the “var” keyword as much as possible. Variables declared using “var” are interpreted more efficiently and will result in faster file generation.
- Module-wide declarations should appear in the following order:
  - variables
  - functions

Function Definitions

- Function arguments do not need to be typed because JavaScript variables are dynamically typed.
- Functions should be created using one of the following styles, based on whether the function belongs to a module or not.
  - **Module functions.** For functions that belong to a module, use the following format:

```
myModule.myFunction = function (argument1, argument2)
{
    var i;

    for (i = 0; i < 10; i++) {/* note the blank line before this */
        argument1 += argument2;
    }

    return (argument1);
}
```

  - **Non-module functions.** When creating functions that do not belong to a module, you should follow this format:

```
function myFunction (argument1)
{
    var x;

    x = argument1;

    return (x);
}
```

- Functions should be defined in alphabetical order.
- There is one blank line between the last declaration in the function body and the first statement in the function.
- Function body braces are aligned with the left margin and the body is indented four (4) spaces.

Comments

- For compatibility with tools currently under development, comments should be formatted as shown in utils.tci. General rules are as follows.
- Border comments begin at column 0 and introduce functions and major sections. Text for a border comment is indented two spaces and starts on the second line. Border comments appear only between function definitions and never within a function body.

```
/*
 *  This is a border comment.
 *  Note the two spaces in from the '*' character.
 */
```

- A source file begins with border comments that display copyright information and identify the file. In some cases, copyright and version information is added to files automatically by source management scripts.

```
/*
 *  Copyright 2002 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *
 */
/*
 *  ======== filename.tci ========
 *  Comment briefly describing what this file contains and how it is used.
 */
```

- All functions have the following style of banner:

```
/*
 *  ======== foo ========
 *  A description of the function is placed here.
 *
 */
```

Note that the name of the function is placed between two strings of eight (8) consecutive "=" characters and a single space. This banner style makes it easy to find all functions declared in a module.

- Multi-line comments are formatted as follows. Note that two (2) spaces follow the asterisk.

```
/*
 *  this is a multi-line indented
 *  comment
 */
```

- Indented comments introduce blocks of code.

```
while (expr) {

    /* this is a simple indented comment */
    if (expr) {
        ‘statement(s)’
    }
}
```

- In-line comments describe individual statements or declarations. All variables should have in-line or possibly multi-line comments giving their meaning.

```
var  frameSize;   /* size of frame in words */
```

- A source file ends with a revision history section in a border comment as shown below. Note that the continuation of a long revision comment is indented by four (4) spaces.

```
/*!
 *! Revision History
 *! ================
 *! 23-Jan-2001 mf: The entire Revision History section must begin with tokens
 *!     that end in an exclamation point, so it can be stripped if necessary.
 *! 24-Dec-2000 mf: Revisions appear in reverse chronological order; that is,
 *!     newest first. The date format is dd-Mon-yyyy.
 *! */
```

Spaces And Tabs

The standard indentation level is four (4) spaces. The tab character (\t) should be treated as eight (8) spaces. Source code should contain either only spaces and no tabs or a mix of spaces and tabs, where a tab is equivalent to eight spaces. Released source code should contain only spaces (no tabs).

Lines should not exceed eighty (80) characters in length. While current editors and UIs allow for much longer lines, most printed material is still eighty characters or less. For publication in manuals, the maximum line length is 74 characters.

Exception Handling

Exception handling code should be used as much as possible. Any code that could generate an error or an exception needs to be written to handle this possibility.

Try and Catch Blocks

Try and catch blocks are used to handle code that may throw an exception. The code that may throw an exception is placed within the try block, and the code that handles the exception is placed within the catch block. When an exception is generated from the try block, it is “caught” in the catch block by storing it as a local variable.

Here is an example:

```
try {
    var x = 20;
    var y = 0;
    var z = x/y;
    print(x + “/” + y + “= “ + z);
}
catch (e) {
    //e is the variable that contains the exception
    print(e);
}
```

There is also another block called finally, which holds code that is guaranteed to execute, no matter what the exception may be. The finally block is where you should place code that you want to be executed, no matter what the outcome of the try/catch block is. The finally block should follow the catch block.

Assertions

Assertions are used to verify that variables and return values are indeed as expected. An assertion could be used to verify that the return value of a function is correct, or that a variable was assigned the correct value. An assert module assert.tci is included with Tconf and should be utilized as much as possible.

Here is an example of how to use assert:

```
/* call the square root function to find the root of 25 */
var x = Math.sqrt(25);

/*
 *  add an assertion to make sure that sqrt works as expected.
 */
assert.add('x == 5');

/* check the assertion just added */
assert.check();
```

References

DSP/BIOS Textual Configuration (Tconf) User's Guide (SPRU007)

C Language Coding Standards for DSP/BIOS and XDAIS (SPRA788)

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