Язык Tconf для DSP/BIOS

В этой статье описывается стандарт языка Tconf, предназначенный для написания конфигурационных файлов для DSP/BIOS и определяющий основные лексические соглашения. Выполнение требований этого стандарта позволяет предотвратить конфликты имен, упрощает использование модулей и увеличивает производительность.
Сторонним разработчикам программного обеспечения предлагается следовать этому стандарту. Однако он носит только рекомендательный характер, в отличие от стандарта eXpressDSP, который обязателен для разработчиков, желающих быть «третьей стороной» — third parties.

Стив КОННЕЛ (Steve CONNELL)
Перевод: Игорь ГУК
gii@scanti.ru

Введение

Язык Tconf используется для конфигурации DSP/BIOS. Он основан на JavaScript. Более подробная информация о языке Tconf приведена в документе "DSP/BIOS Textual Configuration (Tconf) User's Guide" (SPRU007).

Стандарт описывает соглашения по использованию скриптов Tconf при конфигурации проекта. Компания Texas Instruments с 1990 года разрабатывает и улучшает этот стандарт. Сторонним разработчикам программного обеспечения рекомендуется ему следовать. При этом исполнение требований стандарта дает следующие преимущества:

- Упрощается применение модулей. При использовании стандарта программисты знают, где найти определения символов, как создать и использовать различные объекты.
- Упрощается переход к новым ревизиям чипов и увеличивается производительность программного кода. Простая структура и понятные соглашения делают рассмотрение кода, созданного другими, более легким.
- Наличие инструментальной поддержки. Если программист придерживается стандарта, ему доступны простые инструменты для выполнения всех необходимых задач.

Модули

Система программного обеспечения состоит из иерархии модулей. Каждый модуль включает в себя набор взаимосвязанных констант, переменных и функций. И хотя DSP/BIOS предоставляет программисту широкий набор предопределенных модулей Tconf, имеется возможность создавать собственные. Так же как и в языке Си, определение модуля Tconf должно содержать интерфейс и выполняемые им функции в одном файле.

Введем несколько терминологических определений, которые будут полезны для понимания модульности проекта:

- **Модуль.** Наименьшая логическая единица программного обеспечения. Модуль должен содержать все свои функции (а также их описание) и переменные в одном файле.
- **Клиентское приложение (клиент).** Программа, которая вызывает функции интерфейса для выполнения прикладной задачи. Это или *.tcf файл, или один из *.tci файлов.
- **Интерфейс.** Набор взаимосвязанных констант, типов, переменных и функций.
- **Реализация.** Исходный текст функций в модуле.

Лексика модуля

Каждый модуль определяется в своем файле, который должен иметь такое же имя, что и модуль. Например, если модуль называется *myModule*, то он должен быть определен в файле с именем *myModule.tci*. Все идентификаторы и функции, определенные в этом модуле, должны иметь префикс с названием модуля. Например, в файле *myModule.tci* могло бы быть примерно такое содержание:

```
myModule.myInteger = 1;
myModule.myString = "foo";

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

Переменные, определенные в пределах модуля, должны иметь префикс с именем модуля для того, чтобы предотвратить коллизии имен. Переменные модуля нельзя создавать при помощи ключевого слова *var*.

Язык *Tconf* (и JavaScript) позволяет объявлять как локальные, так и глобальные переменные. Чтобы создать локальную переменную, необходимо использовать ключевое слово *var*, и, кроме того, переменная должна быть объявлена в теле функции. Например, в следующем кодексе только *myModule.var_c* является локальной переменной:

```
/* 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;

}
```

Переменные Tconf, определенные в пределах модуля и вне тела функции, всегда являются глобальными переменными. Если глобальная переменная *foo* определена в *myModule.tci* и позже еще раз определена в другом файле, тогда *foo* из *myModule.tci* будет переписана.

Локальные переменные, такие как *myModule.var_c* из предыдущего примера, можно определять без префикса с именем модуля.

Создание модуля

Файлы модуля Tconf должны включать следующее содержание (контент):

1. **Баннер.** Каждый модуль содержит баннер с комментариями, которые описывают модуль.
2. **Объект модуля (объект).** Модуль должен начинаться с создания объекта модуля с таким же именем, что и у самого модуля. Объект модуля всегда нужно создавать вначале, его нельзя включать в алфавитный список переменных. Например, объект модуля в файле *myModule.tci* должен быть создан самым первым, до создания чего бы то ни было в этом файле:

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

Если у модуля есть какие-нибудь внутренние функции, «внутренний» объект, содержащий эти функции, должен быть объявлен сразу же после объекта модуля. Дополнительная информация о внутренних функциях приведена в п. 6.

3. **Объявление переменных.** Все переменные модуля должны быть определены и перечислены в алфавитном порядке после создания объекта модуля. (Заметим, что локальные переменные, определенные в пределах функций, не включаются в этот список.)
4. **Объявление функций.** Все функции должны быть определены в алфавитном порядке после последнего определения модульных переменных.
5. **Функция init.** У всех модулей должна быть функция *init*, которая явно вызывается в конце интерфейса модуля. Это требуется, даже если модуль не нуждается в функции *init*. В этом случае создается пустая функция *init*. Например, функция *init* в *myModule.tci* может быть примерно такой:

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

В конце файла модуля функция *init* должна быть вызвана примерно так:

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

6. **Внутренние функции.** Если в модуле определяется функция, которая является внутренней по отношению к модулю, то эта функция должна быть определена через «внутренний» объект. Таким образом, если необходимо создать внутреннюю функцию, сначала создается внутренний объект, который должен быть определен сразу после создания объекта модуля:

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

Внутренние функции должны быть определены через «внутренний» объект следующим образом:

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

Функции, которые определены как внутренние, как правило, создает проектировщик модуля. Эти функции обычно используются для специальных операций. Внутренние функции не должны использоваться за пределами модуля, в котором они созданы.

Вот пример файла *myModule.tci* с определением модуля *myModule*:

```
/*
 *  myModule.tci
 */

/* Определение объекта модуля myModule */
var myModule = {};

/* Определение переменных модуля в алфавитном порядке */
myModule.myInteger = 1;
myModule.myString = "hello";

/* Определение функций модуля в алфавитном порядке */
myModule.foo = function()
{
}

myModule.init = function()
{
}
myModule.xyz = function()
{
    /*
     *  Эти переменные не должны объявляться выше
     *  в алфавитном списке, потому что они являются
     *  локальными в данной функции
     */
   var x;
   var y;
   var z;
   x = y = z = 2;
}

/* Явный вызов функции init */
myModule.init();
```

Исходные файлы модуля

Приведем листинги одного *.tcf и двух *.tci файлов, взятые из стандартных примеров для DSP/BIOS. Изучите структуру этих файлов, ее мы объясним в следующем разделе, где показано, как создать собственный скрипт Tconf.

Файл TCF: hostio.tcf

В начале этого файла находятся строки, в которых импортируются файлы TCI:

```
/*
 *  Авторские права с 2004 года принадлежат Texas Instruments Inc.
 *  Все права защищены. Собственность Texas Instruments Inc.
 *  Права на использование, дублирование или раскрытие этого
 *  кода ограничены, код предоставляется только по контракту.
 *  @(#) DSP/BIOS_Examples 5,0,0 05-03-2004 (biosEx-a16)
 */
/*
 *  ======== hostio.tcf ========
 *  Пользовательский скрипт конфигурации является общим
 *  конфигурационным файлом большинства примеров
 */

/*
 *  Импортирование настроек dsk6713, которые являются
 *  общими для большинства примеров.
 */
utils.importFile("dsk6713_common.tci");

/*
 *  Импортирование конфигурационных настроек hostio,
 *  которые относятся ко всем платформам.
 */
utils.importFile("hostio.tci");

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

Файл TCI: dsk6713_common.tci

```
/*
 *  Авторские права с 2004 года принадлежат Texas Instruments Inc.
 *  Все права защищены. Собственность Texas Instruments Inc.
 *  Права на использование, дублирование или раскрытие этого
 *  кода ограничены, код предоставляется только по контракту.
 *  @(#) DSP/BIOS_Examples 5,0,0 05-03-2004 (biosEx-a16)
 */
/*
 *  ======== dsk6713_common.tci ========
 *  Специальные настройки только для платформы dsk6713.
 */

/*
 *  Настройка платформозависимых карты памяти, частоты CLK и т. д.
 */
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
};

/*
 *  Настройка выбранной платформы с параметрами,
 *  определенными выше.
 */
utils.loadPlatform("ti.platforms.generic", params);

/*
 *  Разрешение общих для всех примеров настроек BIOS
 */
bios.enableRealTimeAnalysis(prog);
bios.enableMemoryHeaps(prog);
bios.enableRtdx(prog);
bios.enableTskManager(prog);

/*
 *  Определение функции инициализации, которую вызывают
 *  до вызова main()
 */
bios.GBL.CALLUSERINITFXN = true;
bios.GBL.USERINITFXN = prog.extern("GBL_setPLLto225MHz");
/*
 *  Включение heap в SDRAM и определение метки SEG0
 *  для использования
 *  heap, если в программе определен useMemSettings.
 */
bios.SDRAM.createHeap = true;
bios.SDRAM.heapSize = 0x8000;
bios.SDRAM.enableHeapLabel = true;
bios.SDRAM["heapLabel"] = prog.decl("SEG0");
```

Файл TCI: hostio.tci

```
/*
 *  Авторские права с 2004 года принадлежат Texas Instruments Inc.
 *  Все права защищены. Собственность Texas Instruments Inc.
 *  Права на использование, дублирование или раскрытие этого
 *  кода ограничены, код предоставляется только по контракту.
 *  @(#) DSP/BIOS_Examples 5,0,0 05-03-2004 (biosEx-a16)
 */
/*
 *  ======== hostio.tci ========
 *  Включаемый файл Tconf, импортируемый файлом hostio.tcf,
 *  в котором настраиваются глобальные платформонезависимые
 *  объекты BIOS, свойства и параметры.
 *
 */

/*
 *  Создание и инициализация двух HST-объектов для входа
 *  и одного для выхода
 */
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");

/*
 *  Разрешить все компоненты DSP/BIOS
 */
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");
/*
 *  Создание и инициализация LOG-объектов
 */
var trace;
trace = bios.LOG.create("trace");

/*
 *  Установить длину для буфера LOG_system
 */
bios.LOG_system.bufLen = 512;
```

Общая структура исходных файлов

В следующих подразделах описано, как должен быть сгруппирован и организован Tconf-код.

Группирование кода

Код Tconf делится на две категории — независимый и зависимый от выбора конкретного типа процессора (платформы) код.

- Файлы с платформонезависимым кодом (сохраняемые в файлах с расширением *.tci) должны содержать код, который является (или может являться) общим для более чем одной стандартной архитектуры (ISA).
  - Эти файлы могут импортироваться при помощи скриптов в файл *.tcf или в другой файл *.tci.
  - Называть эти файлы рекомендуется в том же стиле, что и С-файлы проекта. Если С-файлы называются в соответствии с их функциональными возможностями, то и файлы с кодом Tconf следует называть так же. Например, файл, который настраивает PIP-объекты DSP/BIOS, можно назвать *myPipes.tci*.
  - Эти файлы должны быть доступны всем ISA, которым они нужны, и сохранять в общей директории для того, чтобы избежать их дублирования.
- Платформозависимые файлы содержат код, который зависит от спецификации ISA, на котором ведется разработка программы. Многие файлы *.tci попадают в эту категорию. Скрипты в файлах *.tcf обычно так же платформозависимы, так как импортируют платформозависимые файлы (однако возможна разработка и платформонезависимых файлов *.tcf с использованием аргументов командной строки).

Замечание. *Для создания конфигурации используются файлы *.tcf, которые импортируют как платформозависимые, так и платформонезависимые файлы *.tci. Таким образом, любая конфигурация производится через файл *.tcf и не может быть выполнена только файлом *.tci.*

Организация кода

Код скрипта *.tcf должен поддерживать определенную структуру, описываемую далее. Если размер кода в группе достаточно большой, он должен быть сохранен в отдельном подключаемом файле Tconf (*.tci) и категоризирован либо как платформозависимый, либо как платформонезависимый (см. раздел «Группирование кода»).

1. Определить все глобальные переменные. Если эти переменные находятся в отдельном файле *.tci, тогда этот файл должен быть импортирован при помощи функции *utils.importFile( )*.
2. Загрузить как можно раньше аппаратные средства поддержки. Для этого используется функция *utils.loadPlatform( )*, которая может быть вызвана следующими способами:
   - **Зависимый синтаксис (рекомендуемый).** При этом способе используется файл *.tci, в котором сначала определяется объект *params*, а затем вызывается функция:

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

   Сам файл *.tci подключается в файле *.tcf. В качестве примера можно посмотреть файлы hostio.tcf и dsk6713_common.tci, листинги которых приведены в разделе «Исходные файлы модуля».
   - **Прямой синтаксис.** В этом случае вызов функции осуществляется непосредственно в файле *.tcf:

```
utils.loadPlatform("Dsk5510");
```

   При этом вызов функции должен быть сделан как можно раньше. Эту форму вызова можно использовать и в файле *.tci, однако в этом случае необходимо файл *.tci импортировать в файл *.tcf как можно раньше. В документе "DSP/BIOS Textual Configuration (Tconf) User's Guide (SPRU007)" описывается применение данной функции для различных платформ.
3. Определить все платформозависимые свойства, объекты DSP/BIOS, переменные и параметры. Если свойства, объекты, переменные и параметры находятся в отдельном файле *.tci, необходимо его импортировать.
4. Определить все платформонезависимые свойства, объекты DSP/BIOS, переменные и параметры. Если свойства, объекты, переменные и параметры находятся в отдельном файле *.tci, необходимо его импортировать.
5. Вызвать функцию *prog.gen( )* в конце файла. Рекомендуется сначала провести проверку наличия ошибок и осуществить вызов функции следующим образом:

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

Хотя в приведенном примере у *prog.gen( )* и нет никаких параметров, разрешается определять в качестве параметра директорию. Если директория указана, то в ней должны находиться файлы конфигурации. Если аргументов нет, файлы конфигурации размещаются в текущей директории. ***Важное замечание:*** *если определяется произвольная директория для конфигурационных файлов, тогда ее местоположение должно быть указано в проекте или make-файле для корректного выполнения процесса компиляции.*

Основные лексические соглашения

В данном разделе приводятся лексические соглашения, относящиеся ко всем исходным файлам Tconf. Вообще, лексические соглашения для Tconf аналогичны лексическим соглашениям языка С. Более подробно данный вопрос рассмотрен в руководстве "C Language Coding Standards for DSP/BIOS and XDAIS (SPRA788)".

Идентификаторы

В JavaScript для переменных тип явно не указывается, потому что используется динамическое определение типов. Это означает, что тип определяется непосредственно во времени выполнения инициализации переменной. Например:

```
var myInteger = 10;                          // целочисленная переменная
var myString = «programming is fun»;  // строковая переменная
```

- Имена переменных и функций состоят из ряда целых или сокращенных слов. Первое слово полностью состоит из строчных букв. Последующие слова в имени обычно начинаются с прописной буквы. Подчеркивание не используется между словами в имени, но может отделять суффикс в конце. Например:

```
var countNumberFrames;
```

- Языковые ключевые слова всегда сопровождаются знаком ';':

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

Дополнительная информация об идентификаторах приведена в разделе «Объявления».

Выражения

Не должно быть свободного места в унитарных выражениях или между первичными операторами и их операндами.

- Пример унитарного выражения:

```
-17
```

- Пример первичных операторов:

```
a[i]
s.member
f(x, y)
```

Свободное пространство может существовать между другими типами операторов и их операндами. Например:

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

Круглые скобки используются для определения порядка вычислений, особенно в выражениях, имеющих несколько вложенных уровней.

Если в языке JavaScript нет явного определения, то порядок вычисления может быть произвольным.

Операторы

Все зависимые операторы закрываются скобками и выравниваются на четыре позиции.

Существует хотя бы один оператор в строке.

Протяженность строки — восемь позиций.

Объявления

Каждая переменная или любой другой идентификатор объявляются на отдельной строке:

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

Переменные можно объявлять и по-другому, так как JavaScript — это язык с динамическим определением типов. Это значит, что тип переменной может быть определен во время выполнения программного кода, и более того, одна и та же переменная может быть использована для хранения данных различных типов. Например, следующий код является правильным:

```
var myInteger = 100; /* myInteger хранит целочисленное значение */

var myInteger = «hello world»;
/*  myInteger хранит уже строковую переменную,
 *  перезаписав предыдущее целочисленное значение. */
```

Рекомендуется использовать при объявлении переменных ключевое слово *var* как можно чаще. Такое объявление переменных позволяет получить более эффективный и компактный программный код.

Объявление модулей должно происходить в следующем порядке:

- переменные,
- функции.

Определение функций

- Аргументы функции объявляются без указания типа, так как в JavaScript используется их динамическое определение.
- При создании функции необходимо учитывать ее принадлежность модулю.
  - **Модульные функции.** Для функций, которые принадлежат модулю, используется следующий аргумент:

```
myModule.myFunction = function (argument1, argument2)
{
      ar i;
 for (i = 0; i < 10; i++) {/*перед оператором пропустить строку */
      argument1 += argument2;
 }
      return (argument1);
}
```

  - **Внемодульные функции.** При создании функций, которые не принадлежат модулю, необходимо придерживаться следующего формата:

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

      x = argument1;

      return (x);
}
```

- Функции должны быть определены в алфавитном порядке.
- Есть одна чистая линия между последней декларацией в теле функции и первым утверждением в функции.
- Тело функции заключается в скобки и выравнивается по четырем позициям.

Комментарии

Для обеспечения совместимости с современными средствами разработки комментарии должны быть отформатированы, как это показано в файле utils.tci. Общие правила следующие:

- Граничные комментарии могут начинаться с нулевой позиции в строке и описывают функции и основные секции. Текст таких комментариев начинается со второй строки и имеет два пробела в начале. Граничные комментарии возможны только между определениями функции, внутри тела функций их использовать нельзя:

```
/*
 *  Это граничные комментарии.
 *  Важное замечание. Необходимо два пробела от символа '*'
 */
```

- Файл с исходным кодом начинается с граничных комментариев, где приводится информация об авторском праве и идентифицируется файл. В некоторых случаях информация об авторском праве и версии добавляется автоматически менеджером исходных скриптов:

```
/*
 *  Авторские права с 2004 года принадлежат Texas Instruments Inc.
 *  Все права защищены. Собственность Texas Instruments Inc.
 *  Права на использование, дублирование или раскрытие этого
 *  кода ограничены, код предоставляется только по контракту.
 *
 */
/*
 *  ======== filename.tci ========
 *  Комментарий с кратким описанием содержания
 *  и предназначения файла
 */
```

- Все функций имеют следующий баннер:

```
/*
 *  ======== foo ========


 *  Здесь размещается описание функции.
 *
 */
```

Небольшое замечание. *Имя функции выделено двумя последовательностями из восьми знаков '=' и размещено на отдельной строке. Такой стиль баннера помогает находить все функции, объявленные в модуле.*

- Многострочные комментарии отформатированы следующим образом. Обратите внимание, что имеется два пробела между звездочкой и началом комментариев:

```
/*
 *  Это многострочный выровненный
 *  комментарий
 */
```

- Выровненные комментарии программных блоков:

```
while (expr) {

   /* Это простой выровненный комментарий */
   if (expr) {
      'statement(s)'
   }
}
```

- Строчные комментарии описывают отдельные операторы или объявления. Все переменные должны иметь строчные или многострочные комментарии, описывающие их предназначение:

```
var frameSize;              /* размер окна в словах */
```

- Файл с исходным кодом заканчивается секцией истории ревизий, оформленной в виде граничных комментариев, как это показано далее. Важно отметить, что перед продолжением длинного комментария на новой строке добавляется четыре пробела.

```
/*!
 *! История ревизий
 *! ================
 *! 23-янв-2001 года: Все комментарии в секции истории ревизий
 *!     должны начинаться с символа восклицательного знака,
 *!     что позволит, в случае необходимости, их удалить.
 *! 24-дек-2000 года: Ревизии перечисляются в обратном
 *!     хронологическом порядке; вначале описывается самая новая.
 *!     При этом используется следующий формат даты:
 *!     ДД-МММ-ГГГГ.
 *! */
```

Количество символов в строке и позиции табуляции

Стандартный размер сдвига — четыре пробела. Символ табуляции (\t) должен быть обработан как восемь пробелов. Исходный код должен содержать или только пробелы и никаких символов табуляции, или сочетания пробелов и символов табуляции, каждый из которых эквивалентен восьми пробелам. Окончательный вариант исходного кода («релиз») должен содержать только пробелы (никаких символов табуляции).

Длина строки не должна превышать восьмидесяти символов. И хотя новые редакторы поддерживают намного более длинные строки, большое количество печатных материалов имеют длину строки 80 символов или мень-

ше. Поэтому для публикации в руководствах используются строки длиной 74 символа.

Обработка особых ситуаций

Код обработки особых ситуаций необходимо, по возможности, использовать максимально часто. Любой код, который может создать ошибку или исключение, должен быть написан с использованием кода обработки особых ситуаций.

Блоки try и catch

Для обработки особых ситуаций используются блоки *try* и *catch*. Программный код, который может создать исключение или ошибку, помещается внутрь блока *try*, а код обработки возникшего исключения помещается внутрь блока *catch*. Когда возникает исключение в блоке *try*, его «захватывает» блок *catch*, сохраняя его как локальную переменную.

Вот пример использования блоков *try* и *catch*:

```
try {
      var x = 20;
      var y = 0;
      var z = x/y;
      print(x + «/» + y + «= « + z);
}
catch (e) {
      //e — переменная, которая содержит исключение
      print(e);
}
```

Существует еще один блок, называемый *finally*, включающий код, который гарантированно выполняется независимо от того, какое исключение произошло. Этот блок следует непосредственно за блоком *catch,* и код, который он включает, выполняется независимо от результата обработки блоков *try/catch*.

Утверждения

Утверждения используются, чтобы проверить правильность значений. Утверждения можно использовать для проверки корректности возвращаемого значения функции или того, что переменной присвоено правильное значение. Модуль с необходимым кодом для реализации функционала утверждений содержится в файле *assert.tci*, его желательно включить в файл Tconf с исходным кодом и использовать максимально часто.

Вот пример того, как можно использовать утверждения:

```
/*  вызов функции извлечения квадратного корня из числа 25 */
var x = Math.sqrt(25);

/*
 *  добавить утверждения для проверки правильности результата.
 */
assert.add('x == 5');

/* проверить добавленное утверждение */
assert.check();
```

Литература

1. DSP/BIOS Textual Configuration (Tconf) User's Guide (SPRU007).
2. C Language Coding Standards for DSP/BIOS and XDAIS (SPRA788).
3. DSP/BIOS Tconf Language Coding Standards (SPRAA67).