T-Engine Forum
Specification

TEF040-S103-1.01.01/en
2006-06-06

T-Format (3): Global Symbol Naming Rule in C Language

T-Engine
Forum

Number:	TEF040-S103-1.01.01/en
Title:	T-Format (3): Global Symbol Naming Rule in C Language
Status:	[ ] Working Draft, [ ] Final Draft for Voting, [X] Standard
Date:	2003/02/28 First Edited
	2003/08/01 Updated to 1.01.00
	2003/10/27 Updated to 1.01.01
	2006/06/06 Updated to 1.01.02

Table of Contents

Foreword

The T-Engine Project has been established to offer an open, real-time standardized development environment with the aim of achieving a ubiquitous computing environment where everything has a computer incorporated in it and is connected to a network. T-Engine is trying to offer an efficient development environment for the development of portable information devices, home electronic appliances and other network devices in a short period of time.

T-Engine employs network security architecture called eTRON whose architecture is intended to prevent tapping, falsification, and disguise of malicious users so that electronic information can be safely delivered to the other party through insecure network channels.

To support efficient development, T-Engine standardizes hardware (T-Engine board) and real-time kernel (T-Kernel), and especially encourages distribution of middleware. T-Engine is also aimed at smoothing cooperation among semiconductor makers, hardware makers, software makers and system manufacturers, encouraging mutual business dealings, reducing development time and cost, thus enabling high value added product offerings in a short period of time. The combination of advanced semiconductors, implementation and software technologies in T-Engine makes it suitable and unrivaled for the development of advanced application products.

Scope

T-Format specifies the code format of middleware and application software that run on T-Engine/T-Kernel. T-Format includes the following three specifications:

1. Source-code style guideline
   Source-code style format for middleware and application that run on T-Engine/T-Kernel. By using the format, source codes written by various vendors can be combined and compiled/linked as one program.
2. Standard binary format
   Standard executable format for distributing, in binary code, middleware and application software that run on T-Engine/T-Kernel. Executable code format and debugger symbol format are defined.
3. Standard documentation format
   Type and format of documentation that is attached to distributed middleware and application software.

This document defines the format and rules for assignment of software vendor codes, which are part of the naming rules of T-Format.

Normative References

[1] T-Engine Forum. ”T-Format (2) : T-Engine Vendor Code System”, TEF-040-S102, 2002.

Terms and Definitions

Middleware Middleware means application software that runs on T-Engine/T-Kernel, or software that provides some kind of service(s) to other middleware.

Note: The term “middleware” is used in the T-Kernel specification as the software module(s) that acts as a “subsystem” (“middleware in a narrow sense”). The term is used in this document in a sense broader than that and includes any software that provides services to application software (“middleware in a broad sense”).

1. T-Engine Middleware Models

The following is middleware architecture models on the T-Engine architecture and middleware classification as breakdown of each model:

1.1 Classification by the Way It Act

Middleware is broken down into the following three types:

(1) Subsystem-Type Middleware
The middleware which acts as a subsystem on T-Kernel. Services are provided to user applications in two ways: (1) the applications call them through the extended SVC provided by the subsystem; (2) the subsystem modules are called when each event happens.

(2) Device Driver-Type Middleware
The middleware that uses device management functionality that T-Kernel provides. You can use this kind of middleware through the general-purpose device driver interface that T-Kernel provides.

(3) Library-Type Middleware
The middleware that acts as a user library. User applications need to be linked with the necessary library in code creation. The applications receive services by calling a function that the library provides or by accessing a global variable that the library provides.

(4) User Task-Type Middleware
The middleware that acts as a user-task server. We call such a server a "'manager server task." User applications receive services from the server task, using the inter-task communications or synchronization functionality that T-Kernel provides.

1.2 Classification by the Service It Provides

Middleware can also be classified by the service it provides. This classification is for convenience, and a service may be able to be classified as more than one type.

(1) Device Driver-Type Middleware
The middleware that provides an API (application programming interface) in a certain level of abstraction concerning with each device on T-Engine or each device connected to T-Engine.

Example: RS232C driver, PCMCIA driver, Ethernet driver, USB driver, Graphic Card driver

Communication devices such as RS232C, LAN NIC, USB, ISO14443, and PCMCIA further connect another device with them. To achieve portability of the device driver which controls a connected device, the API of the lower-level communication devices will be standardized (see "chapter 5: Standard Middleware Specification").

Because processing of devices such as LAN NIC, storage media, and printer is complex, they also tend to equip another interface of high abstraction level on them (and the complexity is further compounded by the fact that there are many types of interface available). In this case, too, the high-level API of the device drivers needs to be standardized. As for devices to provide general functionality, the API will be standardized as necessary.

(2) Abstract Device-Type Middleware
The middleware that provides software interface of high abstraction level to single or plural devices on or connected with T-Engine.

Example: TCP/IP, Printing I/F, FAT file system, GUI package (window system)

Abstract device-type middleware is basically built on device driver-type middleware or another piece of abstract device-type middleware. Therefore the assumed low-level API needs to be in line with the standardized high-level API of the device driver-type middleware or the high-level API of the other piece of abstract device-type middleware. The high-level API that abstract device-type middleware provides will be standardized if the API has generality or it is possible for the API to be called from other middleware.

(3) Extended Kernel-Type Middleware
The middleware that provides services of higher abstraction level through T-Kernel or the middleware functions to the hardware and software resources that T-Kernel manages.

Example: Process management function, virtual memory function, loader function, CLI (Command Line Interpreter) function.

(4) Middleware to Provide General Functions
The middleware that provides general calculation functions, especially the ones that have tenuous relations with the resources provided by the kernel or external devices.

Example: Sort function, search function, random number generation function, encryption/authentication function

2. Global Symbol

To guarantee that pieces of middleware in T-Format can be combined for use, this section sets out the "global symbol rules." The major part of the rules is aimed at avoiding malfunctions in compiling/linking a middleware program, which consists of programs that two or more vendors wrote, in C language, combined into one. The rules define, in particular, the way to use the globally available name space that library-type middleware provides.

2.1 T-Engine Vendor Code

The middleware vendors that provide middleware compliant with T-Format will receive their own T-Engine Vendor Code[1]. Their TVC will be included in a global symbol to avoid the same symbol being used by two or more vendors.

Example

Vendor name:	YRP Ubiquitous Networking Laboratory
Domain name:	unl

2.2 File Name of Middleware

The following rules will be applied to the name of the files constituting a piece of middleware (the binary file and header file of a library, etc.) complied with T-Format in order to guarantee uniqueness within the T-Engine Forum. (See also 2.6)

```
<Library file name> ::= "lib"<Middleware name> "." <Extension>
<Other file name> ::= <Middleware name> "." <Extension>
<Middleware name> ::= <Software vendor name> "_" <Functionality name> ["_"
<Detail name>]
```

Functionality Name

A functionality name shall express the functionality that the file of the middleware provides. Each functionality name must be unique and not be used by two or more vendors.

A functionality name shall be an alphanumeric character ("a"-"z" and "0"-"9") string of two to eight characters.

Example

Functionality:	MPEG2 CODEC
Functionality name:	mpeg2

Detail Name

Detail name is used to discern each binary file/header file if a piece of middleware is made of more than one binary file/header file. A detail name, like a functionality name, shall be an alphanumeric character string of two to eight characters (out of "a"-"z" and "0"-"9").
Example of a middleware file name
File name of library-type middleware for MPEG2 CODEC developed by the vendor whose vender code is "unl":

```
libunl_mpeg2.a
unl_mpeg2_basic.h
```

Name of Functions to Export

The rule for naming a function that is included in a library file of a piece of middleware and exported to application is as follows:

<Functionality name> ::= <Middleware name> "_" <Functionality identifier>

The <Functionality identifier> is a alphanumeric character string of two to eight characters (out of "a"-"z" and "0"-"9") and indicates the name of functionality of the function.
Example

mpeg2 encoding function:	`void unl_mpeg2_encode(...);`
mpeg2 decoding function:	`void unl_mpeg2_decode(...);`

2.3 Name of Global Variables to Export

The rule for naming a global variable that is included in a library file of a piece of middleware and exported to an application is as follows:

<Variable name> ::= <Middleware name> "_" <Variable identifier>

The <Variable identifier> is an alphanumeric character string of two to eight characters (out of "a"-"z" and "0"-"9") and indicates the name of functionality of the variable.
Example
The variable, etronid, used in the mpeg2 library is expressed as follows:

```
char unl_mpeg2_etronid[16];
```

2.4 Name of Data Type Used for Functions and Global Variables

The rule for naming data structures used for arguments or return values of functions, or for global variables, both of which are to be exported, is as follows:

```
<Data structure name> ::= <Middleware name> "_" <Data structure identifier>
```

The <Data structure identifier> is an alphanumerical character string of two to eight characters (out of "a"-"z" and "0"-"9") and indicates the functionality name of the variable.

Example

```
struct point {
  int x;
  int y;
};
typedef struct point unl_mpeg2_point;
```

2.5 Name of Constant Macros Included in Header Files

The rule for naming constant macros to define special values assigned to arguments or return values of functions to export, or special values set up for global variables to export is as follows:

```
<Macro name> ::= <Middleware name(capital letters)> "_" <Macro identifier>
```

Example

```
#define UNL_MPEG2_MAXID 225
#define UNL_MPEG2_MINID 5
```

2.6 Recommendation for Use of Alias

The above naming rules are for avoiding malfunctions in code generation where you use more than one piece of middleware together, which are developed by two or more vendors independently. The rules, however, contain the following difficulty applying:

1. The name would be long.
2. The name would be different from the one generally known.

As we just explained, the naming rules are for avoiding malfunctions in combining one vendor's middleware with another vendor's, and you do not need to apply them when you use a piece of middleware alone. To, then, eliminate the above difficulties, we recommend you to use an alias for a macro, etc. that is shorter and easier to understand.

For example, if you applied the naming rules to socket-interface middleware of TCP/IP, the function name would be:

```
unl_tcpip_socket();
```

We recommend you to develop a macro set with a following alias:

```
#define socket unl_tcpip_socket
```

The following rules will be applied to the name of the header files for alias definition.

<header file name for alias definition> ::= <Middleware name> "_common." <Extension>

Example

unl_mpeg_common.h

```
#include "unl_mpeg.h"
#define DREQ UNL_MPEG_DREQ
#define decode(x) unl_mpeg_decode(x)
```

unl_mpeg.h

```
typedef struct {
...
} UNL_MPEG_DREQ;
extern int unl_mpeg_decode ( unl_mpeg_dfmt *);
```

3. Middleware Package Name

You can give a package name to your T-Engine middleware. A middleware package name indicates a characteristic or a feature of the middleware, and a package name is assumed to be used for package search, etc. in a software development environment that handles middleware. The rule for naming middleware packages is as follows:

```
<Package name> ::= <Middleware name> ”.” <Extension>
<Middleware name> ::= <Software vendor name> “_” <Functionality name> [“_”
<Detail name>] “_” <Version representation>
```

The <Functionality name> and <Detail name> shall be the same as the ones used for the middleware file name. The <Version representation> will be defined in the next section.

Example: An example of middleware package naming:

The package name of the library-type middleware for MPEG2 CODEC provided by UNL is as follows:

```
unl_mpeg2_codec_010311
```

4. Version Representation

The version of a piece of middleware shall be alphanumerical characters of five digits as below:

```
xyyzz
```

The xx, yy, and zz represent the following:

x: 0-9: major version number

The number will increment when a version change comes with removal of backward compatibility of the middleware. Therefore change in the major version number of a middleware means the software using the middleware might require its source code changed.

yy: a0-z0, 00-99: minor version number

Where the API of a piece of middleware changes but its backward compatibility remains, the major version number of the middleware will be the same, while the minor version number will increment.

Change in the minor version number of a piece of middleware means the software using the middleware will work after re-linking. a0-z0 will be used for pre-releasing, and 00-99 will be used for official releases. For example, the alpha release number for major version 1 will be:
1.a0.zz
The beta release number for major version 3 will be:
3.b0.zz

zz: 00-99: minor release number

Version change for a piece of middleware, while keeping both its API and compatibility in logical operations seen from outside, shall be made by incrementing the minor release number with no change made to the major version number and minor version number. This applies to bug fixing or implementation of a more efficient algorithm, etc. Change in the minor version number of a piece of middleware means the software using the middleware will work after re-linking. In principle, the version number of middleware released for the 1st time will be **1.00.00**. However, if a piece of middleware released for the 1st time is ported one for T-Engine from existing software, the version number of the middleware does not have to be **1.00.00**.