T-Kernel Integrated Development Environment (IDE) for Seamless Middleware Development

Min ZOU   Jai GAY Choon Chye   Chu Lih KWEK

Abstract
Recently, there has been growing interests and demands in adopting a new real-time operating system, T-Kernel for embedded system solutions. In order to help customers/developers to adopt T-Kernel easily, this article presents a solution by integrating the T-Kernel interface libraries into the Renesas HEW (High-performance Embedded Workshop). Several practical examples are illustrated to show that the proposed solutions can accelerate T-Kernel based middleware and application software development.

1. Introduction

T-Kernel is a new generation of open source real-time operating system (RTOS) inherited from μITRON [15], which runs on billions of microprocessors. It has good potential to become an industry standard embedded operating system in near future. Currently most of Renesas middleware and application software are designed for μITRON. However, porting those middleware to T-Kernel becomes cumbersome due to the difference between T-Kernel’s and Renesas middleware’s de-facto tool chains. Therefore, there is a strong need for a proven, robust and easy-to-use development environment for T-Kernel embedded software development. In this article, we present a new T-Kernel development environment solution, which can benefit both Renesas customers and embedded developers in the following ways:

- To shorten the turn-around time for supporting customers who are interested in both T-Kernel and existing Renesas middleware solutions (e.g., migrating from μITRON to T-Kernel)
- To help customers achieve cost-saving and better standardization by utilizing open source T-Kernel based solutions
- To minimize the quality assurance effort by employing reliable and proven tool chains for T-Kernel core and Middleware
- To ease the developer’s steep learning curve of adopting a new RTOS like T-Kernel

We begin in section 2 by introducing the motivation behind the work and describing the current issues. Section 3 explores the problems faced in the environment construction and presents the solution. Section 4 illustrates a few case studies, before we conclude with a summary in Section 5.

2. Motivation and current situation

In this section, we start by highlighting the key advantages of adopting T-Kernel. Then we access the Renesas middleware availability, followed by a discussion on the issues of T-Kernel de-facto tool chain and development environment.

2.1 Advantages of adopting T-Kernel

T-Engine is a standard architecture, developed by T-Engine Forum [17], for next-generation, real-time embedded systems aimed at improving software productivity for these systems. T-Kernel is the real-time operating system (RTOS) under the architecture. It provides the following advantages:

- *Strong standardization*
  The standardization includes kernel system call interface, the guidelines for the hardware execution, device driver model, etc. It also maintains a single unified source code governed by T-License [19]. All these assure middleware portability at source level among different platforms, which is a key advantage for users to have freedom of not being dependent on specific CPU vendor(s).
- *Hard real-time performance*
  T-Kernel is a high-performance RTOS based on ITRON whose real-time performance is one of the best [18]. The task dispatch time in the T-Kernel is in the sub-microsecond range [3].
- *Dynamic resource management*
  Memory resource is dynamically managed; automatic assignment of ID numbers of kernel objects, such as tasks and semaphores, facilitates dynamic addition/loading of middleware and device driver.
- *Support of CPUs with/without MMU*
  T-Kernel can take advantages (e.g., memory protection) of MMU (memory management unit), which is usually included in modern high-functionality embedded processors. It can also operate on CPUs without MMU.
- *Scalability into a more sophisticated system*
  T-Kernel is equipped with a mechanism called subsystem, which enables kernel extension (e.g., T-Kernel Standard Extension) and the construction of both lightweight and advanced systems on the same kernel [7].

In this article, we focus on T-Kernel software development on SuperH based T-Engine. Nonetheless, our solution can be applied to other CPU cores supported by T-Kernel.

2.2 Vast array of Renesas Middleware

One of our primary goals is to allow customers/developers to benefit from both features provided by T-Kernel and reuse of existing Renesas middleware, which widely spread over audio/video, imaging, speech and networking solutions. Most of them run on μITRON and are developed with Renesas tool chain (C/C++ compiler).

Although there are many other compilers (like open source

GNU tool chain), developers still prefer a single tool chain throughout one project, because binary codes generated by different C/C++ compilers (even they support the standard object format, like ELF) may not be fully compatible with each other. Conversion from one tool chain to another is sometimes a very painful process. Besides, porting to a new compiler may incur some performance penalty. Therefore, customers of existing Renesas middleware would prefer to stick to Renesas tool chain environment.

2.3 Current issues with T-Kernel de-facto development environment

The standard development environment proposed by T-Engine Forum is called T-Builder [7], which comprises a collection of tools from GNU development system. As shown in the Fig 1, it consists of GNU C compiler, GNU assembler, GNU linker, GNU debugger front-end, and various GNU binary tools. It usually runs on a Unix-based operating system, which connects to the target T-Engine system via a serial cable for remote cross-platform debugging and program download purposes.

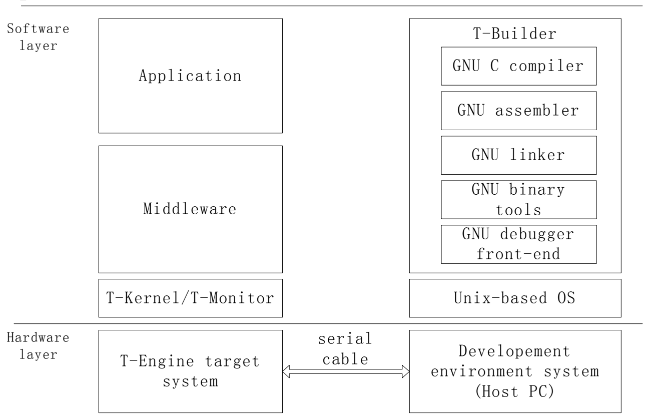


Fig 1. T-Engine standard development environment

The T-Builder environment employs a set of open source tool chains, which is very cost-effective. However as many of toady's embedded software projects are complex and time-critical, such an environment may not be sufficient and efficient compared to the integrated development tool with a scaled debug solution [2]. The High-performance Embedded Workshop (HEW) is an advanced integrated code development and debugging environment (IDE) for developers of embedded applications using Renesas microcontrollers and microprocessors. In the following we describe the issues with the default T-Kernel development environment, and highlight the counter-measure in HEW.

2.3.1 Compiler issues

The default GNU tool chain recommended by T-Engine Forum is the cross GCC (version 3.0.4) for SH targets in the T-Engine Development Kit [13] (hereinafter referred to as "GCC"). The default tool chain for SH targets in HEW IDE is SH Compiler (hereinafter referred to as "SH-C"). SH-C is an optimizing C/C++ compiler for the SH family. SH-C has many advantages over GCC as shown in Table 1.

In terms of optimization, GCC optimizes code at C/C++ language level to achieve portability among different CPU architectures; while SH-C performs further CPU-dependent optimization by utilizing CPU-specific features, such as cache-savvy (e.g., prefetch instruction) and pipeline optimization [6]. In addition, SH-C also provides a set of intrinsic functions [5] for accessing CPU resources (e.g., FPU registers for high-performance matrix operations in SH4).

Table 1 GCC vs SH-C feature comparison

	GCC	SH-C
Full support of Embedded C	Yes	Yes
Full support of Embedded C++ [10]	No	Yes
Full support of DSP-C [11]	No	Yes
Full support of FPU	No	Yes
Development Roadmap for SH famly	No	Yes
License	GPL [12]	Commercial
Technical support	Fair	Good

2.3.2 Debugger issues

In T-Builder, the debugging tool is console-based GDB (GNU Debugger) to do remote debugging over serial cable (Fig 3), which is a primitive method. HEW IDE provides On-Chip Debugger (OCD) or In-Circuit Emulator (ICE) with fully graphical interaction. OCD such as Renesas E10A utilizes on-chip resources such as JTAG and a hidden debug monitor to offer a cost effective source code debugging solution. It connects to host PC through USB connection; provides ultra-fast download speed, broad compiler compatibility, and integrated flash programming. All these features are crucial for high-performance debugging purpose.

2.3.3 Customers and developers' preference

Many of the existing Renesas customers and internal developers in Renesas are very familiar with HEW IDE. If they have to learn a new development tool, the development time will increase significantly. According to EE Times report [1], most of developers in Asia still prefer commercial compilers and IDE for current and future use.

3. The T-Kernel IDE solution

Based on Section 2, we take the following considerations in our solution approach:

1. T-Kernel core is fully built and tested using T-Builder.
2. Most existing Renesas middleware are developed and tested using HEW IDE. Some of them require a RTOS support (e.g., μITRON).
3. HEW IDE is an advanced embedded system development tool to shorten the time-to-market and improve productivity, which is favored by most Renesas customers and developers.

We propose to integrate a subset of T-Kernel source code into HEW IDE to enable the speedy software development for T-Kernel while alleviating the need of quality reassurance of compiler reliability, etc. In order to achieve this, we study the T-Kernel source code architecture, identify the porting part, address the porting issues and finally package up the system.

3.1 Open source T-Kernel architecture

T-Kernel source code can be obtained from T-Engine Forum website [17] upon a free registration. At the time of writing, the latest version is 1.01.00. The entire source code is about 72000 lines. Around 82% of them are written in C language and the rest (hardware-dependent portion) are in assembly language. The following table shows the top-level directories included in T-Kernel source code.

Table 2 Directory structure of the T-Kernel

kernel	T-Kernel body (core)
lib	Library
include	Various definition files (Header files)
config	Configuration files including Rominfo
etc	Scripts for make rules, etc.

The kernel directory includes the kernel initialization portion, the core of kernel functions (T-Kernel/OS), the system manager (T-Kernel/SM) and CPU-dependent portion. Normally, code in kernel directory is kept intact unless T-Kernel needs to be ported to a new CPU type.

The lib directory contains the source code of libraries used for the T-Kernel and user program. There are 5 separate modules under lib directory, as shown in Table 3:

Table 3 Libraries in the T-Kernel

libsvc	T-Kernel system call interface library
libtk	T-Kernel standard function library
libtm	T-Monitor system call interface library
crt	Startup routines
libstr	String operation utility library

The **libsvc/libtm** contains the entry routines (written in GNU assembly language) for all the T-Kernel/T-Monitor system calls and some native T-Kernel Extended Service Calls (SVC) [9]. The interface library is CPU-dependent. The two libraries are linked with user program for making T-Kernel/T-Monitor system calls. The rest directories (**libtk**, **crt** and **libstr**) contain the libraries used by both T-Kernel core and user program, such as memory allocation, hardware-dependent timer functions, etc. The **include** directory contains various header files for C language written user programs and T-Kernel core itself. The **config** directory contains configuration files used for the T-Kernel and the T-Monitor. They are hardware-dependent.

For T-Kernel-based program (classified as middleware and application software in Fig. 1) are those programs making use of T-Kernel functionalities through T-Kernel system calls, such as **tk_cre_tsk()** which is used to create a task. In order to build such program, we only need to link the user program object code with the T-Kernel system call interface library, namely **libsvc** (similarly for **libtm**). We demonstrate how it works in the following example.

```
       .text
       .balign 2
       .globl  Csym(tk_cre_tsk)
       .type   Csym(tk_cre_tsk), @function
Csym(tk_cre_tsk):
       mov.l   fno, r0
       trapa   #TRAP_SVC
       rts
       nop

       .balign 4
fno:    .long   TFN CRE TSK
```

Fig 2. Code listing of tk_cre_tsk.S for SH7727

3.1.1 Example of T-Kernel system call flow

Assume we create a T-Kernel-based program (for SH7727 CPU) in which the **tk_cre_tsk()** system call is used. Then, all the T-Kernel system call symbols can be resolved by linking with the **libsvc** library. For example, the file tk_cre_tsk.S (for SH7727) is shown in Fig 2.

Each T-Kernel system call has the same entry routine as the above except that a unique negative number is assigned. For example, **tk_cre_tsk()**'s corresponding number is TFN_CRE_TSK (0x80010100), which indicates the function code and the number of parameters [4]. The Fig 3 shows the flow of a T-Kernel system call.

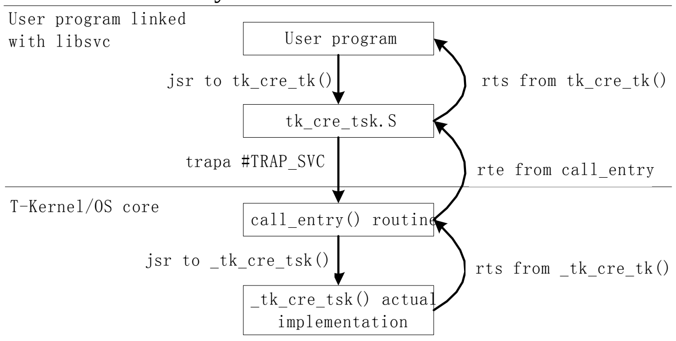


Fig 3. Example flow of a T-Kernel system call

As we can see above, with the interface library **libsvc**, and other libraries in the lib directory (see Table 3), we can use a completely separate building environment (namely, separate tool chain) for middleware/user application from that of T-Kernel core. Note that those native T-Kernel Extended SVC, such as **tk_get_smb()**, are invoked in a similar way as T-Kernel system calls. And their interfaces are also included in the library **libsvc**.

3.2 Addressing the porting issues

According to previous analysis, we only need to port source code under **lib** and **include** directories (see Table 2) to HEW IDE, while leaving the rest (**kernel** and **config**) in original GNU environment. With this strategy, we can reduce the porting work to a minimum level and keep a trusted building environment for T-Kernel core at the same time.

The porting work is essentially to address the compiler differences between GCC and SH-C. The keywords and semantics specified in ANSI C specification are usually not sufficient, so normally many C compilers provide additional features or extensions to the ANSI C specification. Both GCC and SH-C are dialects of ANSI C compiler, so incompatibility mainly lies in those non-ANSI C features (such as inline function) as well as assembly code syntax. For example, GCC provides some "alternative keywords" [8] such as, "__inline__" and "__asm__", which are not recognized by other C compilers. Similarly, SH-C provides language extension through "#pragma" directive [5]. For assembly code, there are more drastic differences in assembler directives because they are usually assembler-specific. In the following subsections, we describe how we resolve the above issues based on SH7727 target.

3.2.1 C inline function

An inline function is a programming language construct used to suggest the compiler to insert the complete body of the function in every context where that function is used. In T-Kernel source, those frequently used and small functions (e.g., **`allocate()`** in lib/libtk/src/memalloc.c) are typically defined as inline functions, in order to eliminate the inherent time overhead in calling those functions, and also to leave more flexibility for compiler to do optimization. To port from GCC to SH-C, the function format has to be changed. For example, the function **`allocate()`** format:

```
Inline VP allocate(QUEUE *aq, size_t size, MACB *macb)
```

is replaced with:

```
#pragma inline(allocate)
static VP allocate(QUEUE *aq, size_t size, MACB *macb)
```

(where Inline is defined as macro for "static __inline__" in T-Kernel). Such replacement applies to all the occurrence of inline functions in the source code.

3.2.2 Inline assembly

In GCC, inline assembly (**__asm__ volatile**) feature allows to embed assembly code in C program. It can appear inside ether normal C function or C inline function. However this feature is not present in SH-C. To achieve the same effect, we create an inline assembly function externally (using **#pragma inline_asm**), and then call it in the C function.

```
Inline UH in_h( INT port )
{
    UH   data;
    /* Compensation for PC card I/O */
    if (((UINT)port & 0xfffc0000) == 0xb8240000
        || ((UINT)port & 0xff3c0000) == 0xb6240000)
        port -= 0x40000;
    Asm("mov.w @%1, %0": "=r"(data): "r"(port));
    return data;
}
```

```
#pragma inline_asm ( in_h_asm )
static UH in_h_asm( INT port )
{
    mov.w    @r4, r0
    extu.w    r0, r0
}

#pragma inline ( in_h )
static UH in_h( INT port )
{
    /* Compensation for PC card I/O */
    if (((UINT)port & 0xfffc0000) == 0xb8240000
        ||((UINT)port & 0xff3c0000) == 0xb6240000)
        port -= 0x40000;
    return in_h_asm(port);
}
```

Fig 4. Code modification for inline assembly

For example **`in_h()`** is an inline function commonly used for memory-mapped hardware register access, in which inline assembly is used. Fig 4 shows how we convert the code.

3.2.3 Assembly code syntax difference

In GCC assembly program, many C language macro directives can still be used, such as #include, #ifdef, #define, etc. However SH-C assembler has the following restrictions:

(1) No C macro directive is allowed;

(2) No C style comment directive (e.g., /* */) is allowed;

(3) Program must be ended with ".end" directive.

For example, Fig 5 shows the ported tk_cre_tsk.S from Fig 2.

```
         .section P_tk, code
         .align   2
         .global  _tk_ref_smb
_tk_ref_smb:
         mov.l    r7, @-r15
         mov.l    r6, @-r15
         mov.l    r5, @-r15
         mov.l    r4, @-r15
         mov.l    fno, r0
         mov      r15, r4
         trapa    #TRAP_SVC
         rts
         add      #4*4, r15

         .align   4
fno:     .data.l  SYSTEM_TK_REF_SMB_FN
         .end
```

Fig 5. Code listing of ported tk_cre_tsk.S for SH-C (SH7727)

3.2.4 Symbol name referencing

SH-C appends an underscore (_) before every symbol (function or variable) declared in C program, while GCC doesn't (when generating ELF binary format). In T-Kernel, **`Csym`** macro is declared to handle the compiler difference.

Furthermore, there is another difference in the method for referencing assembly program symbol names in C programs. In SH-C, to reference an external symbol name (preceded by an underscore (_)) declared in assembly program, in C program, the symbol name must be referenced with removing the leading underscore [5]. However, it is contrary in GCC. For example, the symbol **`_init`** and **`_fini`** declared in the assembly program: lib/crt/crt0/src/sysdepend/std_sh7727/crti.S, are referenced in the C program: lib/libtk/src/startup_elf.c. Thus, those underscores are dropped in the C program for porting to SH-C.

3.2.5 Mimicry of linker script

GCC uses linker script (a text file) to specify section information (including ROM to RAM mapped sections); while in HEW IDE, a GUI configuration tool is used. In the original startup assembly program (in crt directory of T-Kernel), it determines the necessity of ROM to RAM data copying by comparing the two symbols __data_org and __data_start, which mark the end of code section and the beginning of data section. To mimic the same behavior in HEW IDE, we create some empty sections for those symbols, and insert them in the proper location of the section configuration. Besides, ROM to RAM mapping is specified for data section using the GUI configuration. This method turns out to work well without altering the original startup routines.

3.3 Porting verification

The same C program may be compiled into different assembly programs by different compilers, so we need to verify if the porting from GCC to SH-C generates any overhead. Here we examine the same example in section 3.2.3. To do this, we create a simple C program using **`in_h()`** inline function, and compile them into assembly program using GCC and SH-C respectively, and finally compare the difference. The program "`void main() {in_h(0xFFFFFFA0); return;}`" is generated into the following assembly programs (with "**`-O2`**" option specified for GCC; "**`-speed`**" option specified for

SH-C):

```
/* generated by GCC */
main:
    mov.l   r14,@-r15
    mov     #-96,r1
    mov     r15,r14
    mov.w   @r1,r1
    mov     r14,r15
    rts
    mov.l   @r15+,r14
```

v.s.

```
; generated by SH-C
_main:
    .STACK   _main=0
    MOV      #-96,R4
    mov.w    @r4, r0
    extu.w   r0, r0
    .ALIGN   4
    RTS
    NOP
    .END
```

Fig 6. Comparison between assembly programs generated by GCC and SH-C

As show above, both GCC and SH-C correctly expand the inline function and inline assembly function in main routine. More importantly, it is clear that there is no extra overhead generated despite our re-arrangement for **`in_h()`** in section 3.2.2. Besides, note that frame pointer (r14) is used in GCC by default to reduce the bookkeeping on procedure call, while SH-C does not use frame pointer.

3.4 Using the T-Kernel IDE solution package

With the T-Kernel interface libraries ported to HEW IDE, once developers study and understand the T-Kernel API (referring to [9]), they can start middleware and application software development based on open source T-Kernel. Upon completion, the compiled object code can be downloaded/flashed into the target device together with the other three pre-built binaries: T-Monitor, RomInfo, and T-Kernel core. The location of each binary must be pre-defined based on the memory map layout. The RomInfo keeps track of the locations of T-Kernel and application program startup routines, so that T-Kernel and application can boot up successfully. Here is a screenshot for the IDE environment.

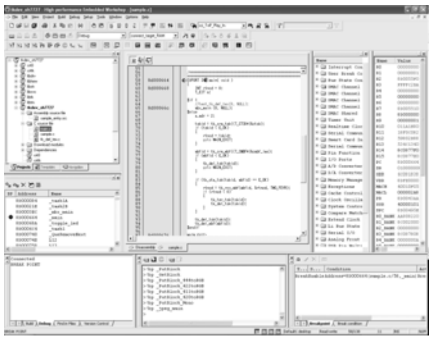


Fig 7. T-Kernel IDE screenshot

4. Case studies

In this section, we illustrate a few projects to show how our proposed solution can accelerate the development process.

4.1 "Plug-n-play" existing middleware without OS intervention

As mentioned in section 2.2, Renesas has a lot of existing multimedia middleware solutions (e.g., JPEG decoder), which are developed using SH-C tool chain under HEW IDE. Most of them do not need to make OS system call directly. Thus, we can easily bring these middleware (in binary form) into our T-Kernel IDE.

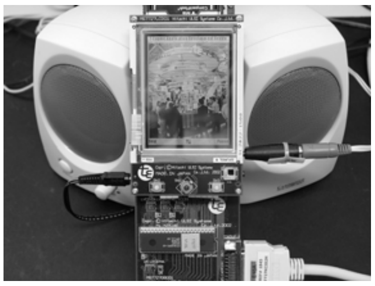


Fig 8. T-Engine/SH7727 media player

This "plug-n-play" concept was exercised in developing a media player using T-Engine/SH7727. In this project, we made use of JPEG and MP3 decoder middleware functions directly in different T-Kernel tasks. The media player is able to play/pause/resume MP3 songs in several bit-rates, and slideshow JPEG images concurrently (see Fig 8 above).

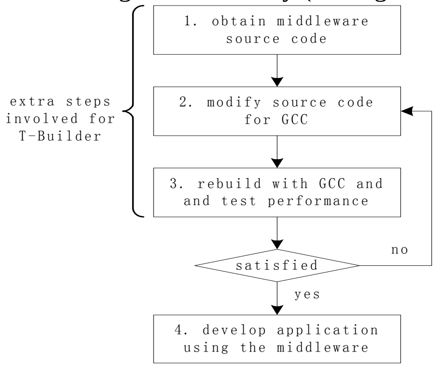

1. obtain middleware source code
2. modify source code for GCC
3. rebuild with GCC and and test performance
satisfied
no
yes
4. develop application using the middleware
extra steps involved for T-Builder

Fig 9. Extra steps for T-Builder

With our proposed T-Kernel IDE, The system prototype was developed rather rapidly (less than one man-month). However, if T-Builder were used, we would need at least another month time to make JPEG and MP3 middleware (with source code provided) ready for GNU development environment, as shown in Fig 9.

While in T-Kernel IDE, the middleware performance has already been assured with SH-C, so we can directly embark on application development. Besides, the integrated debugging environment using E10A via JTAG greatly improved the debugging efficiency.

4.2 Porting existing middleware with OS intervention

As mentioned in section 2.2, many Renesas middleware are developed on top of ITRON. With availabiliy of source code, and understanding of the differences between ITRON and T-Kernel, developers can gracefully port middleware from ITRON to T-Kernel using our T-Kernel IDE solution. The key differences between ITRON and T-Kernel are that: (1) system call APIs are not fully compatible with each other; (2) subsystem and device driver models are used only in T-Kernel; (3) memory management is performed using MMU in T-Kernel.

In order to provide a file system for the above media player project, we ported the UltraFile file system middleware (including the CF card driver) from μITRON to T-Kernel successfully within one week. The ported code is built again with the trusted compiler SH-C, so the file system performance is guaranteed.

4.3 Developing new T-Kernel based middleware

Our T-Kernel IDE solution is a fully functional development platform for creating new T-Kernel middleware. It also helps us in supporting some collaboration projects with customers or partners. Two examples are given here.

A TCP/IP stack middleware was developed using the T-Kernel IDE, by an internship student who had no prior RTOS development experience. He spent about 3-4 months in porting an open source TCP/IP stack called lwIP [16] to T-Kernel and developing an Ethernet driver for T-Engine LAN extension board, while studying T-Kernel API at the same time. This shows that the T-Kernel IDE can help developers to adopt T-Kernel easily and accelerate the development cycle.

We also co-work with partners of T-Engine Application Development Centre[20] to enable wireless sensor networking between several nodes using SH7145 μT-Engines [7] (nT-Engine and pT-Engine platforms were not released at the time of development). T-Kernel was chosen in the project due to its features like compactness, real-time performance, dynamic resource management, etc, and more importantly the middleware reusability for future pT-Engine, which requires ultra-low power consumption. With T-Kernel IDE, we helped the partners (three engineers) kickoff the development work easily. Finally they completed the middleware development within one month.

5. Conclusions

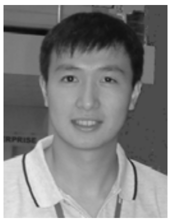
Despite the presence of GNU development environment, T-Kernel based software development remains challenging to many Renesas customers and developers, because: (1) few existing Renesas middleware are built with GNU tool chain; (2) GNU development environment still has issues in terms of compiler features and support, debugging and industry preferences, etc. In section 3, after carefully analyzing the T-kernel source architecture, we present a new solution to integrate T-Kernel interface libraries into Renesas HEW IDE for efficient T-Kernel software development. With several case studies, we proved that T-Kernel IDE solution could help customers port existing Renesas middleware easily and create new middleware/application software for T-Kernel in a short period of time. Furthermore, the friendliness and modular design of the T-Kernel IDE also encourage more developers to participate in T-Kernel software development activities.

Acknowledgement

We thank Mr. Akiya Fukui and Mr. Yoshihiro Chuta Saito (from RSO) for providing the T-Kernel patch for SH7145; Mr. Takao Kobayashi (from Hitachi Software Engineering) for helping in GCC and SH-C comparison; Mr. Shinya Kanda, Mr. Hiroyuki Yamada and Ms. Chie Yamatoya (from RSO) for their support on T-Engine; Mr. Osamu Yamashiro, Mr. Hwee Hwa Low and Mr. Bao Anh Tran Nguyen (from RSS) for their comments on the draft.

References

[1] EE-Times Asia & Gartner Dataquest, “Embedded System Development Trends: Asia”, Feb 2005.
[2] M. Goodchild, "Using the High-performance Embedded Workshop for Embedded Software Application Development", RSO Giho, vol. 2, pp.113-119, March 2005.
[3] J. Krikke, “T-Engine: Japan’s Ubiquitous Computing Architecture Is Ready for Prime Time”, IEEE Pervasive Computing, vol. 4, no. 2, pp. 4-9, 2005.
[4] Personal Media Corporation, “T-Monitor/T-Kernel Implementation Specification for SH7727”, ver. 1.B0.05, 2005.
[5] Renesas Technology, “SuperH™ RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor, Complier Package V.9.00, User’s Manual”, Rev. 1.00, 2004.
[6] Renesas Technology, “SuperH™ RISC engine C/C++ Compiler Package, Application Note”, Rev. 2.00, 2005
[7] K. Sakamura, N. Koshizuka, “T-Engine: The Open, Real-Time Embedded-Systems Platform”, IEEE Micro, vol. 22, no.6, pp. 48-57, 2002.
[8] R. M. Stallman, GCC Development Community, “Using GCC: The GNU Compiler Collection Reference Manual”, the Free Software Foundation, 2003
[9] T-Engine Forum, “T-Kernel Specification”, ver. 1.B0.01, 2005,
[10] http://www.caravan.net/ec2plus/
[11] http://www.dsp-c.org/
[12] http://www.gnu.org/licenses/licenses.html
[13] http://www.personal-media.co.jp
[14] http://www.renesas.com
[15] http://www.sakamura-lab.org/TRON/ITRON/home-e.html
[16] http://www.sics.se/~adam/lwip/
[17] http://www.t-engine.org
[18] http://www.t-engine.org/en/maker/T-EngineFAQ.txt
[19] http://www.t-engine.org/T-Kernel/lisence_e.html
[20] http://www.t-engine.com.sg



Min ZOU
He joined Renesas System Solution Asia in 2004, and is currently an engineer in OS & SH group. Currently he is mainly handling RTOS related development work and SuperH core support. He received B.Sc and M.Sc. degrees in computer science from National University of Singapore.

Jai GAY Choon Chye
He joined Renesas System Solutions Asia in 2003, and is currently a senior engineer in OS & SH group. He was previously doing middleware development. Currently he is mainly handling RTOS related development work and SuperH core support. He received B.Eng. degree in computer engineering from Nanyang Technological University.

Chu Lih KWEK
He joined Hitachi Micro Systems Asia in 1998. He was in Application, Debugger, Emulator, Embedded OS development. Currently, he is the manager for the OS & SH Group including Automotive. He received B.Eng. degree in electrical (electronic) engineering from Nanyang Technological University.