A Universal Real-time Kernel Based on the µITRON Specification

Katsuhito Fukuoka
Akira Yokozawa
Kiichiro Tamaru

Advanced Microcomputer Systems
Engineering Department
Semiconductor System Engineering Center
TOSHIBA Corporation, 589-1, Horikawa-cho, Saiwai-ku, Kawasaki, 210 JAPAN

Abstract

The growing variety of target processors complicates the design of applications. To reduce this complexity, a universal design environment is desired. As the first step in designing the hardware independent environment, here we propose a universal real-time kernel specification for 8 to 32-bit microprocessors. The µITRON specification can be applied to various processors but to specify the universal specification, some decisions need to be made. We will discuss the variations in the word length, trade-offs in hiding architectures, and other related issues. We will then propose the universal specification based on the µITRON specification.

1 Introduction

The µITRON specification was originally proposed as a general supervisory program for low-level microcontrollers and microprocessors. It provides for real-time and multi-tasking operating system functionality on various low-level processors, and increases programming productivity.

Because there were no standard operating systems in the field of microcontroller applications, the µITRON specification was accepted more widely than it was originally anticipated; not only for the applications of the 8-bit processors but also for the higher-level 16-bit and 32-bit processors.

The µITRON specification is widely accepted because it is highly adaptable. To facilitate high efficiency on a variety of processors, the µITRON specification is not rigidly standardized. In other words, its specification is loosely defined so that each implementor can define a detailed specification based on it.

The decisions on the final definition are left to each implementor. This policy is based on the philosophy of "weak standardization"[1] of the TRON project.

However, due to the popularity of the µITRON specification, it is being applied to an increasing number of processors, and users are now confounded with many kernels having similar but slightly different specifications. From the viewpoint of productivity, portability, and education; more rigid standardization is desired. This is the motivation of our project.

Our objective is to increase the level of standardization by providing a source-level compatibility based on the µITRON specification. It will result in the following:

- Hardware independent programming.
- Application portability.
- Standardization of programming style.

When these results are established the user can program without considering the processor architecture and choose a suitable processor after the programming has been completed, or easily change the processor later on, as shown in figure 1.

![Figure 1: Application portability](image-url)
On the other hand, the more strict the specification is, the more difficult it is to maintain optimum performance on various processors.

Attacking this problem, we attempt to define a universal real-time kernel specification with the following guidelines:

- Implementable on a wide variety of processors.
- Conforms with μITRON interface.
- Maintains performance on every processor.

In this article we will discuss the definition of the universal specification in the following sequence.

1. Undefined specifications in μITRON (sec. 3).
2. Policies of our specification (sec. 4).
3. Definition of the universal specification (sec. 5).
4. Evaluation of the universal specification (sec. 6).
5. Evaluation of μITRON (sec. 7).

This discussion is based on the μITRON specification Ver.2.02.01.00[2].

### 2 Previous works

Before our work, the μITRON specification was implemented on various processors independently, and some reports [3], [4], [5] discussed the definition of their specifications.

And there are some implementation reports [7], [8], [9] of the ITRON2 specification[6] for the higher-level processors.

Having learned much from these implementations, we would now like to go further. We now attempt to define the universal specification adaptable to 8 to 32-bit processors.

### 3 Undefined portions in the μITRON specification

When implementing the μITRON specification, we must make decisions regarding the "implementation dependent" portion in the specification. This part is classified as the part where implementors can add their own flavor of performance and functionality in their real-time kernels. This aspect of the μITRON specification encourages competition among implementors.

In our view, the implementation-dependent items are divided into two parts:

- The hardware dependent part.
- The software dependent part.

The former part depends on the architecture of the target processor, while the latter depends on the function of the kernel.

In this section, we would like to clarify the undefined portions in each part. These are discussed later in section 5.

#### 3.1 Hardware dependent part

The most serious problem in defining the universal specification is the variation in architectures ranging from low-level 8-bit processors to high-level 32-bit processors. There are variations in word lengths, interrupt handling, data representations, etc. In addition to these variations, we treat the variation of assembly languages in this section, because the variation is based on the difference of the instruction sets.

### Word length

In the μITRON specification, system call interfaces are defined like the following:

```c
ER ercd = sta_tsk(ID tskid);
```

where the types of the error code ER and the identifier ID are not tightly defined. They are implied as being integer numbers, but their length are not clarified in the specification.

Basically there are two ways of handling these types for 8 to 32-bit processors as shown in figure 2. One is to fix the data length throughout the specification (a), and the other is to scale it according to the word length of the processor (b).

### Interrupt handling

To maintain efficiency in interrupt handling, interrupt architectures are not standardized in the μITRON specification. This part is the most difficult to standardize. The architectures differ in the following ways:

- Number of interrupt vectors.
- Interrupt mask and interrupt level.
- Interrupt enable/disable control bit.

These differences should be treated in the universal specification to maintain portability.
Timer interrupt

The timer interrupt is the only part which should explicitly be left as a hardware dependent specification. At least the interrupt vector number and the interrupt level of the timer interrupt, if necessary, should be reserved for each kernel.

Data representation

There can be differences in data representation such as "endian". But in the view of programming interface, they are not problem because such differences are hidden by programming languages. This problem does not have to be considered in the single processor system, but it will be considered when we extend the specification to the multi-node system using multiple processors in different architectures.

Assembly language interface

In assembly language programming, the different languages must be used because each processor has its own instruction set. In the μITRON specification, the system-call interface is not defined in an assembly language level. But for the convenience of programmers, we would like to define the system-call interface even in assembly languages.

3.2 Software dependent part

By defining the software dependent part in the μITRON specification, distinguishing features can be added to each real-time kernel.

Definition of objects and handlers

Tasks, synchronization/communication objects and handlers are defined statically under the μITRON specification. This means that they are defined in the source program or other configuration files, and generated before execution.

The most basic way to define these objects and handlers is to edit the control data of the kernel directly. But the kernel data is so difficult to read and modify that the resulting code is often quite buggy when using this method. To reduce complexity, some configuration tools should be provided.

System-call set

Generally system-calls are provided as a library. In the μITRON specification the set of the system-call library can be defined.

The set of the system-calls should be defined suitable for both low-level to high-level applications. It should not be too heavy for low-level kernels but should be enough for high-level kernels.

C language interface

In the μITRON specification all system-calls are defined as function prototypes in C language. But types of TIME, TMO and TMSG are left ambiguous.

The types of TIME, and TMO are absolute and relative time, which are used as follows:

\[
\text{ER ercd = wai_tsk(TMO tmout);} \\
\text{ER ercd = set.tim(TIME time);} \\
\text{ER ercd = get.tim(TIME time);} \\
\text{ER ercd = def.cyc(HNO cyhno,} \\
\text{FP cychdr, UINT cyhact, TIME cytime);} \\
\text{ER ercd = def.alm(HNO alhno,} \\
\text{FP almhdr, TIME cytime, UINT tmmode);} \\
\]

Generally, time should be treated as 48-bit data. It is easier to handle when they are defined as UW (32-bit unsigned integer), however it can easily overflow within about 49 days. This can be a problem, even in small applications.

Next is TMSG, a message packet, which heavily depends on the implementation. They are used as follows:

\[
\text{ER ercd = snd_msg(} \\
\text{ID mbxid, T.MSG *pk.msg);} \\
\text{ER ercd = rcv_msg(} \\
\text{T.MSG **pk.msg, ID mbxid);} \\
\]

\[
\]
Because the messages are treated as queue elements in the μITRON specification, they should include a certain amount of pointer area.

Tuning parameters

The kernel should be tuned by the users to be suitable for their applications. For example, the interrupt stack size should be defined according to the complexity of the interrupt handlers in an application; it should also be small enough to minimize RAM consumption. These kinds of turning facilities can be used by the implementor of the real-time kernel to create a competitive advantage.

4 Policies of the universal specification

We set three policies in the definition of the universal specification.

4.1 μITRON fully compliant

We not only satisfied the definition of μITRON specification, we also took into consideration the de facto standards in the industry. For example, the set of system-calls is not dictated by the μITRON specification, but most of the implementation are level 3 compliant. We tried to keep compatibility with former kernels of such implementations.

4.2 C Source-level compatibility

The most important aim of the universal specification is source level compatibility, which provides application portability beyond differences of processor architectures. We tried to define the detailed specification with C language interfaces.

4.3 Applicable for 8 to 32-bit processors

The most significant differences between processors are the following:

- Differences in absolute performance.
- Differences in architecture.

There are differences in performance indexes between low-level and high-level processors. For the low-level processors, because of their low absolute performance, the through-put should be concentrated on, while the response time should be minimized for the high-level processors. Both through-put oriented and response oriented kernels can be derived from the universal specification.

The difference is architectures can be hidden with the virtual processor approach. But we should not take this approach because of the performance requirement. If there is a difference which is hard to hide, we do not hide it and instead make it accessible to the programmer writing the application.

5 Definition of the universal specification

Based on the policies described in section 4, we discuss the implementation-dependent items shown in section 3 and define the universal specification.

5.1 Word length

The data types should be defined according to the characteristics of the data. The task priority TPRI is the easiest one. Because we can assume that the range of TPRI must not exceed the range of 8-bit integers, it can easily be defined as B(8-bit integer).

But some parameters such as the identifier ID and the handler-number HNO cannot be fixed as B because the number of objects and handlers in typical applications exceed the range of the 8-bit integer type. On the other hand, even 32-bit applications do not need 32 bits for ID and HNO, so we define them as H(16-bit integer). However, 16-bit data cannot be passed effectively through registers in 8-bit processors. Furthermore, it is not economical to store 16-bit control data in the restricted memory space. Thus, in 8-bit processors they should be treated as 8-bit data, whose range will be sufficient for 8-bit applications. We also treat the error-code ER in the same manner.

Other parameters as shown below should be scalable according to the word length of the processor:

- Object and handler attribute.
- Wake-up count.
- Semaphore count.
- Event-flag pattern. etc.

They should be long enough for complex applications on high-level processors, and can be short for smaller applications in low-level processors. So they are represent as a UNIT type (scalable integer of 8, 16, and 32-bit). The definitions are summarized in table 1.
Table 1: Type definition

<table>
<thead>
<tr>
<th>type</th>
<th>meaning</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPRI</td>
<td>task priority</td>
<td>B</td>
</tr>
<tr>
<td>ER</td>
<td>error code</td>
<td>H (B for 8-bit)</td>
</tr>
<tr>
<td>ID</td>
<td>object identifier</td>
<td>H (B for 8-bit)</td>
</tr>
<tr>
<td>HNO</td>
<td>handler number</td>
<td>H (B for 8-bit)</td>
</tr>
</tbody>
</table>

As a result of the scalable definition, some types have different ranges among processors. To show the limitations of data types, specification constants are defined as shown in table 2. By referring to the specification constants, application programs can know the limits of each type. For example, the maximum limit of a semaphore count, the word length of an event flag, etc. In this way, application programs can adapt to its running environment and improve their portability.

Table 2: Specification constants

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR_WUP_CNT</td>
<td>maximum of wake-up count</td>
</tr>
<tr>
<td>TR_SUS_CNT</td>
<td>maximum of suspend count</td>
</tr>
<tr>
<td>TR_SEM_CNT</td>
<td>maximum of semaphore count</td>
</tr>
<tr>
<td>TR_FLG_PTN</td>
<td>maximum of event-flag pattern</td>
</tr>
<tr>
<td>TR_INT_MIN/ MAX</td>
<td>interrupt handler range</td>
</tr>
</tbody>
</table>

5.2 Interrupt handling

The interrupt handler is the most response sensitive part of a real-time kernel. Because of performance considerations, this part must not be virtualized. Ironically, this part is deeply affected by the architecture.

First of all, the range of interrupt numbers varies depending on the processor, so we decided not to standardize this part. But to maintain the portability of application programs, we define the specification constants TR_INT_MIN and TR_INT_MAX which indicate the range of the interrupt handler numbers (see table 2). By referring to these constants (for example TR_INT_MIN, TR_INT_MIN+1,...), interrupt handlers can be defined in a portable way.

There are also differences in interrupt mask/level and interrupt enable/disable control. We decided to simplify this architecture with the following two system-calls:

ER ercd = dis_int(void);
ER ercd = ena_int(void);

In the µITRON specification, these system-calls have a parameter of interrupt number (intno), but because this parameter depends on the processor, we omitted it to keep compatibility. As a result, interrupts are disabled/enabled all together by these system-calls.

Finally, regarding interrupt mask and level, we recommend that they are not used for portability. For users who must use this facilities, we provide the following system-calls:

ER ercd = chg_ims(UINT imask);
ER ercd = ins_sts(UINT* p_imask);

Please note that those system-calls are not in the universal specification.

5.3 Timer interrupt

To access the reserved number for the timer interrupt vector, which is used by the kernel, the constant CINT_TMR is defined. It has a predefined value for each kernel, but can be changed in an application program if necessary. The predefined or changed value is used to bind a timer interrupt routine.

In addition to this, the interrupt tic CTIC_TMR (in units of milliseconds) and the timer enable flag CUSE_TMR are defined. When the CUSE_TMR is defined as 0, the timer modules are not bound.

These parameters are treated as tuning parameters described later (see table 5).

5.4 Assembly language interface

For the convenience of programmers, we define system-call macros for assembly languages as shown in table 3. Parameters are not specified as part of the macro invocation because of efficiency considerations. Parameters are passed by registers. The allocation is defined with virtual registers which have the names PAR0, PAR1, ..., PAR4. The macros are used like in the following:

```c
#include <ITRON.inc>

MOV $1,R1  ; task ID
sta_task
```

where ITRON.inc is a header file containing definition of sta_task.

In defining the parameter allocations, pointer data should be treated carefully especially in low-level processors. For example, 16-bit addresses are used in
Table 3: Assembly language interface

<table>
<thead>
<tr>
<th>system-call</th>
<th>PAR1</th>
<th>PAR2</th>
<th>PAR3</th>
<th>PAR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>slp.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wup.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wa1.sem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sig.sem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcv.msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prcv.msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and.msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pget.blk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel.blk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get.tid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get.ver</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dis.int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ema.int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ext.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chgpri.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rot.rdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sus.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ram.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>can.wup.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wa1.flg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pol.flg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set.flg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clr.flg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set.tim</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get.tim</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wa1.tsk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>def.cyc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>act.cyc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>def.alm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tsk.mts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tsk.id</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tsk.pri</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tsk.stat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Parameter register assignment

<table>
<thead>
<tr>
<th>processor name</th>
<th>register name (register length [bit])</th>
</tr>
</thead>
<tbody>
<tr>
<td>A27</td>
<td>PAR0 (32) PAR1 (32) PAR2 (32) PAR3 (32) PAR4 (32)</td>
</tr>
<tr>
<td>B16</td>
<td>PAR0 (16) PAR1 (16) PAR2 (32) PAR3 (32) PAR4 (32)</td>
</tr>
<tr>
<td>C8</td>
<td>XWA (32) B (8) C (32) XDE (32) XHL (32)</td>
</tr>
<tr>
<td>D8</td>
<td>WA (16) B (8) C (16) DE (16) HL (16)</td>
</tr>
<tr>
<td>E8</td>
<td>A (8) B (8) C (16) DE (16) HL (16)</td>
</tr>
</tbody>
</table>

The typical assignment of the virtual registers to real registers are shown in table 4.

5.5 Definition of objects and handlers

In the universal specification we recommend users to write all programs using the C language. In µTRON-based programming, objects and handlers are defined but they cannot be treated as objects in the C language. For example, a task is written as follows:

```c
TASK task() {
    ....
}
```

where the type of TASK is typedef'd as void. So the task is created as an ordinary void function in C language.

It must be defined as a task with some additional data such as a task priority, a task attribute and task stack size.

This information can be written in several ways. But if we add new notation for this purpose, it makes the specification complex.

Considering this, when we use a variable flag in C language it is written as follows:

```c
int flag;
flag = 1;
```

The flag is defined before it is used in the same file with the same language. It seems quite natural. Similarly, a semaphore with an identifier of 10 should be used as follows:

```c
CRE_SEM(10);  // Here the semaphore is defined with the configuration macro CRE_SEM. We used this method in 5 macros as listed in table 6.
```

Because they are written in the C language, there is another merit:
Here the semaphore name EX is used both in the configuration macro and in the semaphore system-call. These macros are expanded during compilation. Most of this translation is handled by the C preprocessor, however because of the limitations of the functionality of the preprocessor, even in ANSI-C, an additional filter (configuration tool) must be used. The flow of compilation is shown in figure 3. The application is compiled as usual in one path. In addition, the configuration data is picked up in a separate path by the configuration tool, and compiled to generate control data. Finally, the control data is linked together with the application program.

Figure 3: Compilation flow

5.6 System-call set

In applying the μITRON specification Ver.2, most implementations [3], [4], [5] conform to the level-3 set. We also comply with this set because it is light yet sufficient for the first implementation of μITRON. We decided to conform to the same set, level-3, with extended timer functions as in level-4. The system-call set is summarized in table 7 using C language conventions.

5.7 C language interface

We already proposed the C language interface in table 7. They are defined by ANSI-C function prototypes.

The absolute time TIME in the μITRON specification is changed to T_TIM to indicate that it is a struct with the following form:

```c
struct t_tim {
    UH utime;
    UH mtime;
    UH ltime;
} T_TIM;
```

where utime, mtime, and ltime are the upper, middle, and lower 16-bit portions of the 48-bit absolute time.

The relative time TMO of waitsk is changed to UW (unsigned 32-bit integer) because it does not have to handle 48-bit time. It is divided into two 16-bit parts of mtmout and ltmout in assembler notation (see table 3).

For managing the message packets, we decided to use a linked list. So the structure of T_MSG is defined as follows:

```c
struct t_msg {
    struct t_msg* pk_msg;
    VB msgcont[];
} T_MSG;
```

where pk_msg is a queue link pointer used by a kernel.

5.8 Tuning parameters

To tune kernels for specific applications, we add tuning parameters in the universal specification, these are listed in table 5.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTK_INT</td>
<td>stack size for interrupt</td>
</tr>
<tr>
<td>CTIC_TMR</td>
<td>interval of timer interrupt</td>
</tr>
<tr>
<td>CINT_TMR</td>
<td>interrupt number for timer</td>
</tr>
<tr>
<td>CUSE_TMR</td>
<td>bind of timer</td>
</tr>
</tbody>
</table>

They are ordinarily C language constants, and are handled by the configuration tool. The default values of the parameters are defined in the header file named ITRON.h together with the definitions of TASK, sta_tsk, etc. Their values can be changed just as the ordinary C constants.
### Table 6: Configuration macro interface

<table>
<thead>
<tr>
<th>name</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRE_TASK</td>
<td>(ID tskid, ATR tskatr, TASKP task, TPRI itskpri, UW stksz);</td>
</tr>
<tr>
<td>CRE_SEM</td>
<td>(ID semid, ATR sematr, UINT isemcnt);</td>
</tr>
<tr>
<td>CRE_MBX</td>
<td>(ID mbxid, ATR mbxatr);</td>
</tr>
<tr>
<td>CRE_FLG</td>
<td>(ID flgid, ATR flgatr, UINT iflgptn);</td>
</tr>
<tr>
<td>DEF_INT</td>
<td>(HNO intno, NATR inhatr, INTHDRP inthdr);</td>
</tr>
</tbody>
</table>

### Table 7: System-call interface (in the form of C language interface)

<table>
<thead>
<tr>
<th>type</th>
<th>name</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER</td>
<td>slp.tsk</td>
<td>(void);</td>
</tr>
<tr>
<td>ER</td>
<td>wup.tsk</td>
<td>(ID tskid);</td>
</tr>
<tr>
<td>ER</td>
<td>wai.sem</td>
<td>(ID semid);</td>
</tr>
<tr>
<td>ER</td>
<td>preq.sem</td>
<td>(ID semid);</td>
</tr>
<tr>
<td>ER</td>
<td>sig.sem</td>
<td>(ID semid);</td>
</tr>
<tr>
<td>ER</td>
<td>rcv.msg</td>
<td>(T_MSG** ppk_msg, ID mbxid);</td>
</tr>
<tr>
<td>ER</td>
<td>prcv.msg</td>
<td>(T_MSG** ppk_msg, ID mbxid);</td>
</tr>
<tr>
<td>ER</td>
<td>snd.msg</td>
<td>(ID mbxid, T_MSG* pk.msg);</td>
</tr>
<tr>
<td>ER</td>
<td>pget.blk</td>
<td>(VP *p_blk, ID mplid);</td>
</tr>
<tr>
<td>ER</td>
<td>rel.blk</td>
<td>(ID mplid, VP blk);</td>
</tr>
<tr>
<td>ER</td>
<td>get.tid</td>
<td>(ID* tskid);</td>
</tr>
<tr>
<td>ER</td>
<td>get.ver</td>
<td>(T_VER* pk.ver);</td>
</tr>
<tr>
<td>ER</td>
<td>dis.int</td>
<td>(void);</td>
</tr>
<tr>
<td>ER</td>
<td>ena.int</td>
<td>(void);</td>
</tr>
<tr>
<td>ER</td>
<td>sta.tsk</td>
<td>(ID tskid);</td>
</tr>
<tr>
<td>void</td>
<td>ext.tsk</td>
<td>(void);</td>
</tr>
<tr>
<td>ER</td>
<td>ter.tsk</td>
<td>(ID tskid);</td>
</tr>
<tr>
<td>ER</td>
<td>chg.pri</td>
<td>(ID tskid, TPRI tskpri);</td>
</tr>
<tr>
<td>ER</td>
<td>rot.rdq</td>
<td>(TPRI tskpri);</td>
</tr>
<tr>
<td>ER</td>
<td>sus.tsk</td>
<td>(ID tskid);</td>
</tr>
<tr>
<td>ER</td>
<td>rsm.tsk</td>
<td>(ID tskid);</td>
</tr>
<tr>
<td>ER</td>
<td>can.wup</td>
<td>(UINT* p_wupcnt, ID tskid);</td>
</tr>
<tr>
<td>ER</td>
<td>wai.flg</td>
<td>(UINT* p_flgptn, ID flgid, UINT waiptn, UINT wfmode);</td>
</tr>
<tr>
<td>ER</td>
<td>pol.flg</td>
<td>(UINT* p_flgptn, ID flgid, UINT waiptn, UINT wfmode);</td>
</tr>
<tr>
<td>ER</td>
<td>set.flg</td>
<td>(ID flgid, UINT setptn);</td>
</tr>
<tr>
<td>ER</td>
<td>clr.flg</td>
<td>(ID flgid, UINT clrptn);</td>
</tr>
<tr>
<td>ER</td>
<td>set.tim</td>
<td>(T_TIM* pk.time);</td>
</tr>
<tr>
<td>ER</td>
<td>get.tim</td>
<td>(T_TIM* pk.time);</td>
</tr>
<tr>
<td>ER</td>
<td>wai.tsk</td>
<td>(W tmout);</td>
</tr>
<tr>
<td>ER</td>
<td>def.cyc</td>
<td>(HNO cyhno, FP cyhdr, UINT cyhact, T_TIM* pk.time);</td>
</tr>
<tr>
<td>ER</td>
<td>act.cyc</td>
<td>(HNO cyhno, UINT cyhact);</td>
</tr>
<tr>
<td>ER</td>
<td>def.aln</td>
<td>(HNO alhno, FP almhdr, T_TIM* pk.time, UINT tmmode);</td>
</tr>
<tr>
<td>ER</td>
<td>tsk sts</td>
<td>(UINT* p_tskstat, TPRI* p_tskpri, ID tskid);</td>
</tr>
</tbody>
</table>
6 Evaluation of the universal specification

We proposed the universal specification in the previous section. In this section we will evaluate the specification from the aspects of (1) adaptability and (2) standardization.

6.1 Adaptability to processor variations

As described in section 3, the major differences between processors are summarized as follows:

- Word length.
- Timer interrupt.
- Interrupt handling.
- Assembly language interface.

For the difference in word length, we decided to use scalable types of INT and UINT. But for some restricted data, such as TPRI and ER, fixed types are used.

This mixed style of type definition will be effective for minimizing control data while keeping compatibility between processors.

For the differences in the timer interrupt and in interrupt handling, we decided not to use them. Instead of hiding differences, we propose that users handle the differences by using the specification constants. This strategy avoids the performance problems of the virtual processor approach.

In the universal specification the interrupt mask is not standardized. If interrupt mask system-calls must be used, we have a recommended convention, but this is not part of the specification. We recommend users to not use this feature to ensure compatibility. This can be a problem for complex applications which require multiple interrupts using interrupt levels.

6.2 Adaptability to low-level processors

In low-level processor applications, throughput is the most important consideration. Kernel functions should not consume a considerable amount of processor resources. For these kinds of applications, the primitive set of system-calls in the universal specification will be preferred. They are sleep and wake-up, and semaphore and mailbox.

In the low-level processor applications, assembly language is still preferred because of the performance requirement. We defined universal interfaces in assembly language for this requirement, but the specification should be extended for other low-level languages such as structured assembly language.

6.3 Adaptability to high-level processors

For high-level processors, the response time is the most important consideration. There are some system-calls which cannot be completed within the limited execution time, such as set, figur, however the interrupt response and task response can be maintained by using a hierarchical architecture [10].

For even higher level applications, to support the complexity of the program, exception management and more functional system-calls should be added. There is no great difficulty with a linear extension of system-calls, but the exception management should be included in the core section of the kernel. It should be carefully designed in the near future.

6.4 Standardization of coding style

In the universal specification, programming style can be standardized in applications for 8 to 32-bit processors. Most of the specifications are summarized in the header file ITRON.h and the application program begins by including this file. In the universal specification, the application program is written with the following items:

- Task.
- Interrupt handler.
- Timer handler.

A skeleton of the program based on the style is summarized in the listing of figure 4.

7 Evaluation of the μITRON specification

We defined the universal specification based on the μITRON specification. In this work, we attempted to adapt the specification for various processors with different architectures, and we did not find any definitions that were too rigid in the μITRON specification. In this sense, the μITRON specification was a good base for the universal specification.

On the other hand, as we described in section 5, the C language interface could have been defined more rigidly in the μITRON specification. We hope that they are specified by using the stricter prototype declaration to improve the standardization level.

The definition of the exception class is not used because the exception handling is not defined in the μITRON specification. Perhaps exception handling
#include <ITRON.h>

#define CSTK_INT stacksize

CRE_TSK(tskid, TA_HLNG, task, tskpri, stksz);
CRE_SEM(semid, TA_NULL, isemcnt);
CRE_MBX(mbxid, TA_NULL);
CRE_FLG(flgid, TA_NULL, fflagpt);
DEF_INT(midno, TA_HLNG, intdr);

TASK task() {
    ....
    def alm(alhno, almhdr, pk_time, tmmode);
    def cyc(cyhno, cychdr, cyhact, pk_time);
    ....
}

INTHDR inthdr() {
    ....
}

TMRHDR almhdr() {
    ....
}

TMRHDR cychdr() {
    ....
}

Figure 4: Skeleton of an application program

in not necessary in the current release of the µITRON specification. But in future extensions, it must include exception handling to support more complex applications.

8 Future work

We have just written the first code of the kernel based on the universal specification. This code is running on a 32-bit processor based on the TRON architecture. We are planning to port the master code to some lower-level processors.

We are also planning extensions to the specification for more applications, both lower and higher level applications. For lower level applications a simple scheduler specification can be delivered. For higher level applications, some extension to the exception handling specification should be considered to improve the stability of the systems.

9 Summary

The universal real-time kernel specification is defined based on the µITRON specification. In this article, we took the implementation-dependent portion from the µITRON specification in section 3 and discussed them in section 4 and 5.

We found that the µITRON specification was a fairy good framework for the design of our specification, but we did find some areas that need improvement (section 7).

We hope our work will contribute to reducing the complexities of real-time programming and will establish the hardware independent design environment.

Acknowledgements

The authors would like to thank Dr. Ken Sakamura of University of Tokyo for his many helpful suggestions, and the members of the TRON Association for their cooperation in standardizing the µITRON specification.

References

