An RTOS allowing the gradual migration

Masahiro Fukuda
Microcomputer Software Dev. Lab.
ULSI Systems Dev. Lab. NEC Corp.
Kawasaki, Japan 211

Hirotaka Nokubi
Common Software Department
NEC Microcomputer Technology, Ltd
Tokyo, Japan 108

Abstract

This paper describes an Real-time operating system (RTOS) which can allow users to achieve the gradual migration from the ITRON [1] kernel to the oITRON [2] kernel. It has some risks for the user to migrate between the two ITRONs because each kernel has a different system call specification. The oITRON promises to speed. The ITRON, on the other hand, provides functionality to make robust programs managing complex application area.

We are successfully developing a new RTOS which has two system call interfaces for the user migration between the two ITRONs with no risk. Our new RTOS allows users to achieve functional migration from the ITRON to the oITRON while both using the ITRON interface and getting nearly the oITRON speed.

1 The oITRON specification feature

A speed oriented design is applied to build the oITRON specification which provides relatively fast system calls. Main strategies in order to get relatively fast system calls are both to eliminate timeout operation from system calls which wait the kernel resources such as the waisem() system call and to restrict the number of resources to be treated by the system calls to one. These strategies speed almost all the oITRON system calls up. No Task Control Block (TCB) is linked into both the timeout queue and the kernel resource wait queue because the oITRON specification does not permit resource wait with timeout option. The oITRON specification reduces the operations to be performed atomically by the system call such as the ter_tsk(), wup_tsk(), and etc. All what the ter_tsk() system call and the others have to perform is just to unlink the TCB from either the kernel resource wait queue or the timeout queue. Never both queues are to be unlinked at any system call.

2 The ITRON specification feature

A function oriented design is applied to build the ITRON Specification which provides timeout options in the system calls such as the wai_flg(), wai_sem(), rcv_msg(), get_blk().
A timeout and exception handling structure is very easy for application programmers to build with the ITRON Specification because the ITRON kernel has timeout option in its system call parameters instead of preparing a handler routine and has the def_exc() system call and the def_ext() system call.

For example, ultrasonic transducer can be used for measuring distance instead of optical rangefinder in the robot system which has fault-tolerancy.

The def_ext() system call provides the user with the means to dynamically define the exception handling routine which is invoked when the system calls return error code except for TE_OK. The control of the task is automatically transferred from the normal part of the program to the exception handling routine which is predetermined by means of issuing the def_exc() system call. The control is returned from exception handling routine to the normal part of the program with error code TE_OK after the exception handling routine manages the exceptional situation. The task continues to run as if nothing were happened.

For example, serial line can be used by the software instead of ethernet media that would be in trouble in order to communicate others when the hardware does not have a just single communication method but has alternative communication methods. For the next example, ultrasonic transducer can be used for measuring distance instead of optical rangefinder in the robot system which has fault-tolerancy.

The def_ext() system call provides the user with the means to define the termination processing routine which is invoked when the task gets termination signal via the ter_tsk() system call invoked by other task or via the abo_tsk() system call invoked by the task itself. The termination processing routine typically returns kernel resources by means of, for example, issuing the sig_sem() system call to return the semaphore resources which is used for mutual exclusion, issuing the rel_blk() system call to release the retrieved memory blocks for other tasks to reuse and so on.

There are two models for exception handling. One is the termination model and the other is the resumption model [3]. The ITRON kernel can apply termination models with ease. The def_ext() system call is used for the termination model and the def_exc() system call is for the resumption model except for retries. The def_exc() system call does not support retries.

The \( \mu \)ITRON kernel, on the other hand, an ap-
An application programmer must statically or dynamically create a handler routine to release a task from the resource wait queue, which issues the system call such as \texttt{waitsem()}, if the programmer would like to emulate timeout facility when the predetermined time is elapsed.

Advantages including easy-to-read, efficient, and being able to avoid increasing size and complexity of the application programs which handle many exceptional and unexpected situations are being able to be provided for application programmers by the ITRON kernel.

The Timeout facility is very useful in I/O programs such as Disk Control System (for example read retry, device not ready situation), Network Control System (for example, receive data error because of collision), which have some exceptional and unexpected situations. The Robot Control System with fault-tolerant programming, for example, must manage real world phenomena. Robot arm may fail to grasp the object to be retrieved because it might get lost some parts of its arm or simply fail to grasp. It is a try and check intensive system because many examinations are needed to decide the next action to be done.

3 Two layered structure

Our new RTOS has two aspects. One aspect is the two layered structure RTOS (the ITRON kernel), which is made of the ITRON interface layer with the \mu ITRON kernel in it, adds the functionality to the user programs. The other aspect is the monolithic RTOS (the \mu ITRON kernel) made of the \mu ITRON core which speeds the user programs.

The \mu ITRON kernel's high speed system calls are available via the ITRON interface layer even though the ITRON kernel is used still system call arguments being compatible with the \mu ITRON kernel's. Figure 2 shows the two layered structure and mixed configuration of the ITRON compatible system calls and the \mu ITRON compatible system calls. For example, the \mu ITRON system call such as the \texttt{waitsem()} can be used by selecting the semaphore which permits just FIFO order, just one resource unit, and just to poll semaphore (the time to be elapsed is equal to 0) and just to wait semaphore forever when the tasks wait semaphores (that is, the \mu ITRON specification compatible semaphores) instead of selecting the semaphore which permits both FIFO order and PRIORITY order, to treat plural resource units, and to wait any time (the time to be elapsed is specified by 32-bit argument) when the tasks wait the semaphores (the ITRON specification compatible semaphore) in the RTOS configuration phase. Either semaphore must be selected by the user.

The System call arguments are shown below even if user would select the \mu ITRON type semaphore.

\begin{verbatim}
err = sig_sem(acc_addrs_of_semaphore, \resource_number);
err = wait_sem(option, acc_addrs_of_semaphore, \resource_number, &time);
\end{verbatim}

The "resource_number" will be forced to 1 by the ITRON interface layer. The "time" argument is forced to 0 if "option" is equal to 1 that means polling request for the semaphore which is the same as the \texttt{preqsem()} system call. The "time" argument is removed if "option" is equal to 0 that means waiting for the semaphore forever.

Our system call argument compatible strategy is very useful and safety. It's useful because all what an application programmer who uses the \mu ITRON compatible semaphore must do in order to speeds his code is just re-configuration the ITRON kernel. It's safety because the migration between the ITRON kernel and the \mu ITRON kernel does not require an extra cost. The user can migrate from the ITRON kernel to the \mu ITRON kernel gradually and safety. The user can migrate back to the ITRON kernel when application program critically needs more functions such as timeout and exception handling facilities than that of the \mu ITRON Specification.

4 Internal structure

This section describes the internal structures, the statically configurable ITRON compatible system calls, and the \mu ITRON compatible system calls.

4.1 Static memory allocation for TCBs and other kernel resources

The memory allocation strategy is drastically changed in our new RTOS. Our previous ITRON kernel's memory allocation strategy is to dynamically retrieve memory in order to create Task Control Blocks (TCB) and other kernel resources which consist of flag control block, semaphore control block, and mailbox control block.

The Static memory allocation strategy is applied, on the other hand, by our new RTOS which retrieves all the memory to create TCBs and other kernel resources when the RTOS starts up. The contiguous
memory region is required to create TCBs, flag control block, semaphore control block, mailbox control block, memory pool control block to manage all the kernel resources efficiently. The Maximum numbers of each kernel resources which are used by the user application programs at run-time are to be informed by the user through our new configurator.

The 16-byte aligned strategy for managing kernel resources by means of using the segment registers of the V30 NEC original Microprocessor which is binary compatible with the i8086 Microprocessor has been applied in our previous ITRON kernel. That strategy is very useful when the ITRON kernel dynamically retrieves kernel resources because it needs only 16-bit to address the TCBs and other kernel resources even though the memory retrieved in order to create the kernel resources might be located someplace in 1M-byte memory space which would need a 20-bit to address as it is.

Our new RTOS has applied, on the other hand, a byte aligned strategy for managing the kernel resources instead of the 16-byte aligned except for the memory management. This strategy combined with the static memory allocation strategy reduced the size of almost all the kernel resources. Because no kernel resources except for memory need the registration queue area, the object identifiers (which are, for example, distinguishable semaphore from flag, mailbox, or memory block). The registration queue areas are used for checking if TE.EXS error is to be returned or not. For example, cre_sem() returns TE.EXS error when the semaphore exists which has the same ID as the "semaphore ID" specified in the cre_sem() parameter. The object identifiers are used for TE.OBJ error is to be returned or not. For example, the wa_sem() returns TE.OBJ error when the specified access address is anything else the semaphore's. Our new RTOS has a limitation that all the kernel resources except for memory must be within 64K bytes. This limitation speeds RTOS codes up because 16-bit register is enough length to address all kernel resources except for memory.

The access address of kernel resources except for the memory control block, which consist of the flag control blocks, the semaphore control blocks, and the mailbox control blocks have the 16-bit offset value in the kernel resource control table segment.

Though the access address of the memory pool control block has also the 16-bit offset value in the kernel resource control table segment, the get_blk() system call return the segment value which is aligned to 16-byte boundary in order to manage 1M-byte memory space with only 16-bit data.
Mixed configuration among the ITRON and µITRON system call routines

Mixed configuration made of both the ITRON compatible system calls and the µITRON compatible system calls is allowed in our new RTOS, for example, which can treat the ITRON compatible flag, the µITRON compatible semaphore, the µITRON compatible mailbox, and the ITRON compatible memory pool as kernel resources still having the same argument interface as the ITRON specification. The same kind of resource management system calls are not allowed to be used in one kernel. For example, both the ITRON semaphore management system calls and the µITRON semaphore management system calls can not used at one kernel.

Not allowed are the µITRON compatible flag management system calls, however, to use because we would like to avoid the confusion of the user. As we would like to always provide the µITRON system calls as faster but less functional than the ITRON system calls for the user, the µITRON compatible flag management system calls is not good because they may be the slow set_flg() system call which unlinks the plural TCBs at one system call. The µITRON specification flag can link the plural tasks into its resource wait queue. On the other hand, the ITRON specification flag does not allow to link plural tasks into its resource wait queue.

Various mixed configuration can be selected. Here is, more detail, in the case given above. The ITRON compatible flag management system calls consisting of cre_flg(), del_flg(), set_flg(), wai_flg(), and flg_adr() and the memory pool management system calls consisting of cre_mpl(), del_mpl(), get_glk(), rel_blk(), and mpl_adr() which have, of course, the timeout option and can manage variable length memory blocks. The µITRON compatible semaphore management system calls consisting of cre_sem(), del_sem(), sig_sem(), wai_sem(), and sem_adr() and the µITRON compatible mailbox management system calls consisting of cre_mbx(), del_mbx(), send_mag(), rcv_mag(), and mbx_adr() which have the same function as those of the µITRON kernel still having the same argument interface as the ITRON kernel. The memory pool management system calls selected above permit the user both to wait for the resource in priority order and to wait for the resource with timeout option because they are conformed to the ITRON specification. On the other hand, both the semaphore management system calls and the mailbox management system calls selected above do not permit the user to wait for the resource in priority order. The semaphore management system calls selected above, moreover, limit the number of resources treated at one system call.

Please note that it is possible to create the µITRON compatible semaphores or mailboxes in this case. It means, for example, that the µITRON compatible sig_sem() system call selected in this configuration releases no more the task which is blocking at the head of the semaphore wait queue in question than the µITRON sig_sem() releases.

4.3 Cyclic wake up and cyclic handler implementation

Although the system calls for cyclical executives are defined in both the ITRON and µITRON specifications, each has a different functionality each other. In the ITRON kernel, the "tasks" are to be waken up cyclically (call cyclic wake up), in the µITRON kernel, the "handlers" are to be invoked cyclically (call cyclic handler). (The handler does not have any context just like the interrupt handler. So, it can't wait on any synchronization primitives like semaphore.) The cyc_wup() system call dynamically registers cyclic wake up request to the task, and the can_cyc() system call cancels it. Note that more than one cyclic wake up requests to one task are allowed. Those wakeup requests to the tasks are counted up on every request until the number of requested count is overflow. The def_cyc() system call registers or cancels cyclic handlers. The act_cyc(), rep_cyc() and ret_tmr() system calls are defined in the µITRON specification for controlling the cyclic handlers. The cyclic handler id number which is arbitrary assigned by the user is used in order to identify the handler.

We implemented cyclic wake up facility by means of using the cyclic handler of underlying µITRON in order to reduce code size and improve maintainability. Figure 3 shows the outline of data structure. We must avoid to make our µITRON kernel slower or bigger, so only one extension that the kernel gives handler's id number as an argument to the handler when the kernel invokes it is applied to the cyclic handler specification to implement cyclic wake up functions. The µITRON specification does not regulate arguments of the cyclic handler. Hence, the cyclic handler defined by the user does not expect to receive any arguments. Then under the µITRON interface is selected, only the ITRON interface layer uses cyclic handler functions, so arbitrary number can be assigned as handler id. We assign index number of control structure of cyclic wake up request as handler.
id. The cyclic handler manages its interval. So, we put task access address and number of times to wake up into control structure of cyclic wake up request. At the time, the handler is invoked with the handler id which is the index number to the structures, then the handler can know the task which must be wake up and the number of times to wake up.

5 Providing application portability

An user oriented design RTOS has been obtained by means of integrating the ITRON kernel and the µITRON kernel. Our RTOS provides the user with a safety migration path from the ITRON kernel to the µITRON kernel. The users can apply, for example, the ITRON kernel running on 16-bit Micro Processing Unit to prototype new products with no migration risk.

5.1 Migration path

Three migration paths are prepared for users after they complete their products which are running on 16-bit Micro Processing Units. The first path is to migrate the µITRON kernel running on 16-bit Micro Processing Unit when the user application programs need less functionality and more speed.

The second path is to migrate the µITRON kernel running on 8-bit Micro Controller Unit when the user application programs need less functionality and a little more speed but need not so high speed as 16-bit Micro Processing Unit.

The third path is to migrate the ITRON kernel running on 32-bit Micro Processing Unit when the user application programs need functionality and much more speed.

The direction of the migration depends on the plan of the user products line, computation power of Micro Processing Unit/Micro Controller Unit, chip cost, etc. The users are promised to migrate any direction including cheap 8-bit Micro Controller Units, high cost performance 16-bit Micro Processing Units, and powerful 32-bit Micro Processing Units.
5.2 Migration support

In order to support user to migrate between ITRONs with ease, we have some recommendations concerned about the declarations of the data type, the names of the structure member, and the argument interface.

5.2.1 The declarations of the data type

We have already provided non-compatible structure declaration. For example, our 32-bit real-time OS RX632, which is conformed with the ITRON1 specification, has the following argument interface.

```c
int chg_pri(void* task_access_address,\n    short new_pri);
```

This is different from the argument interface of that of our 16-bit real-time OS RX116/136/320/423 which has the following argument interface.

```c
int chg_pri(int task_access_address,\n    short new_pri);
```

Our 8-bit real-time OS RX78K which has the following argument interface.

```c
int chg_pri(unsigned int task_id,\n    char new_pri);
```

We can fill this gap by providing some files which consist of many typedef declarations. For example, user application program can have the following sentence at the top of it.

```c
#ifdef NECITRON32
typedef void* T_ACCS ;
typedef short T_PRI ;
#else NECITRON16
typedef int T_ACCS ;
typedef short T_PRI ;
#else NECITRON8
typedef unsigned int T_ACCS ;
typedef char T_PRI ;
#endif
```

The system calls above can be rewritten as follows.

```c
int chg_pri(T_ACCS task_access_address,\n    T_PRI new_pri);
```

5.2.2 The names of the structure member

There may be various names of the structure members because no specification about the naming conventions of the structures' member exists. It is possible, however, to fill the gap among the different makers' ITRON kernels by using C preprocessor. We can provide the following macro sentences for example.

```c
#ifdef NECITRON32
#define CRETSKPKDATA(x) (x).p_data
#define CRETSKPKDATAP(x) (x)->p_data
#else NECITRON16
#define CRETSKPKDATA(x) (x).taskds
#define CRETSKPKDATAP(x) (x)->taskds
#else NECITRON8
#define CRETSKPKDATA(x)
#define CRETSKPKDATAP(x)
#endif
```

5.2.3 The arguments interface

There exist more system call arguments in the ITRON specification than those of the pITRON specification because the ITRON specification has a timeout option, and ability to treat plural number of resources. It is very important to fill this gap, although it is not easy to carry out. The following "WAISEM" system call is the sample.

```c
#ifdef NECITRON32
#define WAISEM(OPT,ACCS,RESNUM,TIME)\
    wai_sem(OPT,ACCS,RESNUM,TIME)
#else NECITRON16
#define WAISEM(OPT,ACCS,RESNUM,TIME)\
    wai_sem(OPT,ACCS,RESNUM,TIME)
#else NECITRON8
#define WAISEM(OPT,ACCS,RESNUM,TIME)\
    wai_sem(OPT,ACCS,RESNUM,TIME)
#endif
```

Unfortunately, it is not always possible to use C preprocessor to automatically replace the upper-case system calls with those of the pITRON because the pITRON system calls have two system calls depend on their context. In other word, the pITRON specification has both the system calls used only on the task context and the system calls only on the interrupt handler context. For example, the sigsem() system call is issued only on the task context and the isigsem() system call is used only on the interrupt handler context.

```c
#ifdef NECITRON32
#define SIGSEM(ACCS,RESNUM)\
#endif
```
It is required to write "ISIGSEM" in the interrupt handling routines in order to use the isig_sem() system call when the name NECITRON8 is defined. The variable "RESNUM" is to be equal to one when the NECITRON8 is defined because the section four provides the way from the \mu ITRON kernel to the ITRON kernel with no risk. The users are to test the adaptability of the applications before migrating from the ITRON kernel to the \mu ITRON kernel. The functionality gap between the ITRON specification and the \mu ITRON specification is considerably reduced.

It is possible to fill the gap between the ITRON arguments interface and the \mu ITRON arguments interface. The interrupt handling routines are, however, to be entirely reconsidered because the interrupt managing routines strongly depend on the microprocessor architectures.

It is very difficult, unfortunately, to fill the gap between the timeout operation of the ITRON specification and the handler of the \mu ITRON specification. To replace the ITRON resource wait system calls with timeout operation as the combination of both the \mu ITRON resource wait system calls without timeout operation and the \mu ITRON handlers which is invoked in some delay time in order to release the task in question from the resource wait queue is not easy to migrate. For example, the ITRON wai_sem() system call with timeout operation must be divided into both the \mu ITRON wai_sem() system call with no timeout operation and the handler routine which will be invoked in some delay in order to issue the \mu ITRON rel_wai() system call against the task which issued the \mu ITRON wai_sem() system call with no timeout operation. This means the application programmers must re-organize the structure of the programs if they would like to migrate from the ITRON kernel to the \mu ITRON kernel even though they still need timeout functionality.

6 Conclusions

An RTOS which allows the user to migrate gradually from the ITRON kernel to the \mu ITRON kernel has been obtained by means of integrating the ITRON kernel and the \mu ITRON kernel. The users can migrate among 8-bit Micro Controller Unit, 16-bit Micro Processing Unit, 16-bit Micro Controller Unit and 32-bit Micro Processing Unit with ease. The migration from the ITRON kernel to the \mu ITRON kernel are especially focused in this paper. No risk migration is provided by our new RTOS which can test the adaptability of the applications in terms of the limited functionality of the \mu ITRON specification.

7 Future work

The hard real-time extension to the ITRON/\mu ITRON kernel is the next main target area that we have currently investigating. Hard deadline is very important concept for replaying voice, music, and motion pictures. The protocol managing priority inheritance of mutual exclusion semaphore is now focused in our laboratory.

Priority inheritance protocol is effective to bound the duration of priority inversion. However, the complex cases described in [4] will also need to be solved. In addition to these cases, therefore both ITRON and \mu ITRON have chg_pri() and ter_tsk() system calls which are functions for changing task priority, terminating task respectively, we will need to consider that how handle chg_pri() or ter_tsk() system calls to the task during it inherits priority.

References

[2] Ken Sakamura, \mu ITRON SPECIFICATION Ver.2.01.00, TRON ASSOCIATION 1989.