RENESAS Tool News

RENESAS TOOL NEWS on August 1, 2007: 070801/tn3

Notes on Using the MR Series of Real-Time OSes

Please take note of the following problems in using the MR series of real-time OSes:

- With using a set_flg or iset_flg service call in combination with ichg_pri
- With using objects of property TA_TPRI
- With clearing an event flag for which two or more tasks waiting

1. Problem with Using a set_flg or iset_flg Service Call in Combination with ichg_pri

1.1 Products and Versions Concerned

M3T-MR32R/4 V.4.00 Release 00 (for the M32R/ECU series, M32R family)
M3T-MR308/4 V.4.00 Release 00 through V.4.00 Release 02, and V.4.00 Release 02A (for the M32C/80 and M16C/80 series, M16C family)
M3T-MR30/4 V.4.00 Release 00 (for the M16C/60, M16C/30, M16C/20, M16C/10, and M16C/Tiny series; M16C family)

1.2 Description

If an interrupt is requested while a set_flg or iset_flg service call is executed, and the invoked interrupt handler issues ichg_pri, the waiting state of the task whose priority has been changed may not be released even if the conditions for releasing the waiting state have been met.

1.3 Conditions

1.3.1 In M3T-MR30/4 and M3T-MR308/4

This problem may occur if the following conditions are all satisfied:
(1) A set_flg or iset_flg service call is issued to handle an event flag of property TA_WMUL.
(2) An interrupt is requested while the service call in (1) is executed, and the invoked interrupt handler issues an ichg_pri service call.
(3) The task from within which the ichg_pri service call in (2) has

been issued joins the queue waiting for the event flag to be handled by the set_flg or iset_flg service call issued in (1).

1.3.2 In M3T-MR32R/4

This problem may occur if the following conditions are all satisfied:

(1) A set_flg or iset_flg service call is issued which takes the ID of an event flag of properties TA_WMUL and TA_TPRI as an argument.

(2) An interrupt is requested while the service call in (1) is executed, and the invoked interrupt handler issues an ichg_pri service call.

(3) The task from within which the ichg_pri service call in (2) has been issued joins the queue waiting for the event flag to be handled by the set_flg or iset_flg service call issued in (1).

1.4 Workaround

Before and after the issuance of the set_flg or iset_flg service call described in Condition (1) in both sections 1.3.1 and 1.3.2, disable and enable interrupts respectively.

1.4.1 Examples in M3T-MR30/4 and M3T-MR308/4

(1) Case when a task has issued a set_flg service call

Do not use the interrupt enable bit, but disable interrupts by changing the processor interrupt priority level IPL to the kernel interrupt masking level (OS interrupt disabled level); then enable interrupts by restoring the level.

```
----------------------------------------
void task(VP_INT exinf)
{
   :
/* Disable interrupt */
#pragma ASM
; Statements below are an example when kernel interrupt
; masking level (OS interrupt disabled level) is 7.
 LDIPL   #7
 NOP
 NOP
 NOP
#pragma ENDASM
 set_flg(ID_flg,0x000000F0);
 /* Enable interrupts */
#pragma ASM
; IPL value is usually 0 when a task is executed.
 LDIPL   #0
 NOP
```

```
  NOP
  NOP
#pragma ENDASM
   :
}
-----------------------------------------
```

(2) Case when the interrupt handler has issued a set_flg service call

Use either of the following ways:

(a) As in (1) above, disable interrupts by changing the processor interrupt priority level IPL to the kernel interrupt masking level (OS interrupt disabled level); then enable interrupts by restoring the level.

(b) Set and clear the interrupt enable bit to enable interrupts and disable them.

```
-----------------------------------------
void inthand(void)
{
   :
/* Example when (a) used */
/* Disable interrupt */
#pragma ASM
; Statements below are an example when kernel interrupt
; masking level (OS interrupt disabled level) is 7.
  LDIPL   #7
  NOP
  NOP
  NOP
#pragma ENDASM
  iset_flg(ID_flg,0x00F0);
  /* Enable interrupt */
#pragma ASM
; IPL value restored to the one before interrupt handler issues
; iset_flg.
  LDIPL   #3
  NOP
  NOP
  NOP
#pragma ENDASM
     :
  /* Example when (b) used */
  /* Disable interrupt */
#pragma ASM
```

```
  FCLR  I
 #pragma ENDASM
  iset_flg(ID_flg,0x00F0);
  /* Enable interrupt */
 #pragma ASM
  FSET  I
  NOP
 #pragma ENDASM
    :
 }
 ----------------------------------------
```

NOTICE:

Insert or remove NOP instructions after the LDIPL, FSET, and FCLR instructions depending on the timing when IPL is changed by LDIPL, and the timing when the change of the I flag made by FSET and FCLR is reflected.

Note that the effects of the LDIPL, FSET, and FCLR instructions vary from MCU to MCU.

1.4.2 Example in M3T-MR32R/4

```
----------------------------------------
void task(VP_INT exinf)
{
    :
 /* Disable interrupt */
 asm(" mvfc     R0,PSW¥n"
    " and3     R0,R0,#0xFFBF¥n"
    " mvtc     R0,PSW¥n");
 set_flg(ID_flg,0x000000F0);
 /* Enable interrupt */
 asm(" mvfc     R0,PSW¥n"
    " or3      R0,R0,#0x0040¥n"
    " mvtc     R0,PSW¥n");
    :
}
----------------------------------------
```

1.5 Schedule of Fixing the Problem

We plan to fix this problem in the next release of the products.

2. Problem with Using Objects of Property TA_TPRI

2.1 Products and Versions Concerned

M3T-MR32R/4 V.4.00 Release 00 (for the M32R/ECU series, M32R family)

M3T-MR308/4 V.4.00 Release 00 through V.4.00 Release 02, and
V.4.00 Release 02A (for the M32C/80 and M16C/80 series, M16C family)
M3T-MR30/4 V.4.00 Release 00 (for the M16C/60, M16C/30, M16C/20,
M16C/10, and M16C/Tiny series; M16C family)

2.2 Description

If a service call is issued to make a task enter the waiting state for an object of property TA_TPRI, and the interrupt handler invoked by an interrupt requested while the above service call is executed issues such a service call that releases the above waiting state several times, the state of the task may incorrectly be changed.
And the task may not join a proper queue, waiting or ready; indefinite values be written to indefinite addresses.

2.3 Conditions

2.3.1 In M3T-MR30/4 and M3T-MR308/4

Two suits of conditions, A and B below, are possible.

(1) Conditions A

This problem occurs if the following conditions are all satisfied:

(a) Two or more tasks join any one of the following queues, each of which is waiting for an object of property TA_TPRI:
- Queue waiting for a semaphore
- Queue waiting for transmission of a data queue
- Queue waiting for transmission of a long data queue *
- Queue waiting for transmission of a short data queue **
- Queue waiting for acquisition of memory of a fixed-length memory pool
- Queue waiting for reception of a mailbox message

NOTES:
*In M3T-MR30/4 only
**In M3T-MR308/4 only

(b) A service call is issued to make a task enter the waiting state for any one of the objects in (a).

(c) While the service call in (b) is executed, an interrupt is requested, and the invoked interrupt handler releases the waiting states of all the tasks that are joining the queue in (a).

Example 1 of Conditions A:
Assume that two tasks are already joining the queue waiting for semaphore ID_sem1 of property TA_TPRI. This meets Condition (a).

```
----------------------------------------
void task1(VP_INT exinf)
```

```
{
    :
  wai_sem(ID_sem1);  /* Condition (b) */
    :
}
/* An interrupt requested while wai_sem executed, and invoked
   interrupt handler inthand executed                    */
void inthand(void)
{
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
}
```

Explanation:
If Conditions A were satisfied, task1 would enter the waiting state for a semaphore and join the queue waiting for it. In the Example 1 above, however, task1 goes to the waiting state for a semaphore, but do not join the queue waiting for it.
So indefinite values are written to indefinite addresses.

Example 2 of Conditions A:
Assume that two tasks are already joining the queue waiting for semaphore ID_sem1 of property TA_TPRI. This meets Condition (a).

```
void task1(VP_INT exinf)
{
     :
  wai_sem(ID_sem1);  /* Condition (b) */
     :
}
/* An interrupt requested while wai_sem executed, and invoked
   interrupt handler inthand executed. In addition, multiple
   interrupts generated during execution of inthand, and
   inthand2 executed                                    */
void inthand(void)
{
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
}
```

```
void inthand2(void)
{
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
}
```

Explanation:
If Conditions A were satisfied, task1 would never enter the waiting state because it acquired a semaphore by the second isig_sem service call issued by interrupt handler inthand2.
In the Example 2 above, however, task1 goes to the waiting state for a semaphore, but do not join the queue waiting for it.
So indefinite values are written to indefinite addresses.

(2) Conditions B

This problem may occur if conditions (a) and (b) are satisfied and either Condition (i) or (ii) is met.

(a) Two or more tasks join a queue waiting for an event flag that has the TA_TPRI, TA_WMUL, and TA_CLR properties.

(b) A wai_flg or twai_flg service call is issued to handle the event flag in (a), where the argument waiptn of the wai_flg or twai_flg service call satisfies no waiting condition.

(i) While the wai_flg or twai_flg service call in (b) is executed, an interrupt is requested, and the invoked interrupt handler issues an iset_flg service call to handle the event flag in (a), and iset_flg sets the bit pattern of the event flag so that the waiting condition of wai_flg or twai_flg can be satisfied.

(ii) While the wai_flg or twai_flg service call in (b) is executed, an interrupt is requested, and the invoked interrupt handler issues two or more service calls to handle the event flag in (a), and iset_flg releases the waiting states of all the tasks joining the queue waiting for the event flag.

Example 1 of Conditions B:
Assume that two tasks are already joining the queue waiting for event flag ID_flg 1 that has the TA_TPRI, TA_WMUL, and TA_CLR properties. This meets Condition (a).

```
void task1(VP_INT exinf)
{
  FLGPTN flgptn;
     :
  wai_flg(ID_flg1,(FLGPTN)0x000F,TWF_ANDW,&flgptn);
                              /* Condition (b) */
     :
}
/* An interrupt requested while wai_flg executed, and invoked
   interrupt handler inthand executed                    */
void inthand(void)
{
     :
  iset_flg(ID_flg1,(FLGPTN)0x000F);    /* Condition (i) */
     :
}
```

Explanation:
If Conditions B were satisfied, task1 would never go to the waiting state because its waiting condition be satisfied by the iset_flg service call issued by interrupt handler inthand.

In the Example 1 above, however, task1 goes to the waiting state for an event flag and joins the queue waiting for it.

Example 2 of Conditions B:
Assume that two tasks, task2 and task3, are already joining the queue waiting for event flag ID_flg 1 that has the TA_TPRI, TA_WMUL, and TA_CLR properties. This meets Condition (a).

```
void task1(VP_INT exinf)
{
  FLGPTN flgptn;
     :
  wai_flg(ID_flg1,(FLGPTN)0x000F,TWF_ANDW,&flgptn);
                              /* Condition (b) */
     :
}
void task2(VP_INT exinf)
{
  FLGPTN flgptn;
     :
  wai_flg(ID_flg1,(FLGPTN)0x00F0,TWF_ANDW,&flgptn);
```

```
                                /* Condition (a) */
    :
}
void task3(VP_INT exinf)
{
 FLGPTN flgptn;
    :
 wai_flg(ID_flg1,(FLGPTN)0x0F00,TWF_ANDW,&flgptn);
                                /* Condition (a) */
    :
}
/* An interrupt requested while wai_flg executed, and invoked
  interrupt handler inthand executed                    */
void inthand(void)
{
    :
 iset_flg(ID_flg1,(FLGPTN)0x00F0);       /* Condition (ii) */
    :
 iset_flg(ID_flg1,(FLGPTN)0x0F00);       /* Condition (ii) */
    :
}
-----------------------------------------
```

Explanation:
If Conditions B were satisfied, task1 would enter the waiting state for an event flag and join the queue waiting for it because its waiting condition be not satisfied by the iset_flg service call issued by interrupt handler inthand.
In the Example 2 above, however, task1 goes to the waiting state for an event flag, but do not join the queue waiting for it.
So indefinite values are written to indefinite addresses.

2.3.2 In M3T-MR32R/4

Two suits of conditions, A and B below, are possible.

(1) Conditions A

This problem may occur if the following conditions are all satisfied:

(a) Four or more tasks join any one of the following queues, each of which is waiting for an object of property TA_TPRI:

- Queue waiting for a semaphore
- Queue waiting for transmission of a data queue
- Queue waiting for acquisition of memory of a fixed-length memory pool
- Queue waiting for reception of a mailbox message

(b) A service call is issued to make a task enter the waiting state for any one of the objects in (a).
(c) While the service call in (b) is executed, an interrupt is requested, and the invoked interrupt handler issues a service call to releases the waiting states of all the tasks that are joining the queue in (a). In addition, the condition for releasing the waiting state of the task for which the service call in (b) has been issued is satisfied.

Example of Conditions A:
Assume that four tasks are already joining the queue waiting for semaphore ID_sem1 of property TA_TPRI. This meets Condition (a).

```
----------------------------------------
void task1(VP_INT exinf)
{
    :
  wai_sem(ID_sem1);  /* Condition (b) */
    :
}
/* An interrupt requested while wai_sem executed, and invoked
   interrupt handler inthand executed                    */
void inthand(void)
{
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
  isig_sem(ID_sem1); /* Condition (c) */
    :
}
----------------------------------------
```

Explanation:
If Conditions A were satisfied, task1 would never go to the waiting state because it acquired a semaphore by the isig_sem service call issued by interrupt handler inthand. In the Example, however, task1 goes to the waiting state for a semaphore and joins the queue waiting for it. The semaphore count, which would be 0, goes to 1.

(2) Conditions B

This problem may occur if the following conditions are all satisfied:

(a) Four or more tasks join a queue waiting for an event flag of properties TA_TPRI and TA_WMUL.

(b) A wai_flg or twai_flg service call is issued to handle the event flag in (a), where the argument waiptn of the wai_flg or twai_flg service call satisfies no waiting condition.

(c) While the wai_flg or twai_flg service call in (b) is executed, an interrupt is requested, and the invoked interrupt handler issues an iset_flg service call to handle the event flag in (a), and iset_flg sets the bit pattern of the event flag so that the waiting condition of wai_flg or twai_flg can be satisfied.

Example of Conditions B:
Assume that four tasks are already joining the queue waiting for event flag ID_flg 1 that has the TA_TPRI and TA_WMUL properties. This meets Condition (a).

```
----------------------------------------
void task1(VP_INT exinf)
{
  FLGPTN flgptn;
     :
  wai_flg(ID_flg1,(FLGPTN)0x000F,TWF_ANDW,&flgptn);
                                  /* Condition (2) */
     :
}
/* An interrupt requested while wai_flg executed, and invoked
   interrupt handler inthand executed                    */
void inthand(void)
{
     :
  iset_flg(ID_flg1,(FLGPTN)0x000F);    /* Condition (i) */
     :
}
----------------------------------------
```

Explanation:
If Conditions B were satisfied, task1 would never enter the waiting state for an event flag because its waiting condition be satisfied by the iset_flg service call issued by interrupt handler inthand. In this example, however, task1 goes to

the waiting state for an event flag and joins the queue waiting for it.

2.4 Workaround

Before and after the issuance of the service call described in Condition (b), disable and enable interrupts respectively.

2.4.1 Examples in M3T-MR30/4 and M3T-MR308/4

Do not use the interrupt enable bit, but disable interrupts by changing the processor interrupt priority level IPL to the kernel interrupt masking level (OS interrupt disabled level); then enable interrupts by restoring the level.

```
-----------------------------------------
void task(VP_INT exinf)
{
  :
/* Disable interrupts */
#pragma ASM
; Statements below are an example when kernel interrupt
; masking level (OS interrupt disabled level) is 7.
 LDIPL   #7
 NOP
 NOP
 NOP
#pragma ENDASM
 wai_sem(ID_sem1);  /* Condition (b) */
/* Enable interrupts */
#pragma ASM
; IPL value is usually 0 when a task is executed.
 LDIPL   #0
 NOP
 NOP
 NOP
#pragma ENDASM
     :
}
-----------------------------------------
```

NOTICE:

Insert or remove NOP instructions after the LDIPL, FSET, and FCLR instructions depending on the timing when IPL is changed by LDIPL, and the timing when the change of the I flag made by FSET and FCLR is reflected.

Note that the effects of the LDIPL, FSET, and FCLR instructions vary from MCU to MCU.

2.4.2 Example in M3T-MR32R/4

```
----------------------------------------
void task(VP_INT exind)
{
   :
/* Disable interrupts */
asm(" mvfc     R0,PSW¥n"
   " and3     R0,R0,#0xFFBF¥n"
   " mvtc     R0,PSW¥n");
wai_sem(ID_sem1);  /* Condition (2) */
/* Enable interrupts */
asm(" mvfc     R0,PSW¥n"
   " or3      R0,R0,#0x0040¥n"
   " mvtc     R0,PSW¥n");
   :
}
----------------------------------------
```

2.5 Schedule of Fixing the Problem

We plan to fix this problem in the next release of the products.

3. Problem with Clearing an Event Flag of for Which Two or More Tasks Waiting

3.1 Products and Versions Concerned

All the versions of M3T-MR32R and M3T-MR32R/4
(for the M32R/ECU series, M32R family)
All the versions of M3T-MR308 and M3T-MR308/4
(for the M32C/80 and M16C/80 series, M16C family)
All the versions of M3T-MR30 and M3T-MR30/4
(for the M16C/60, M16C/30, M16C/20, M16C/10, and M16C/Tiny series, M16C family)
All the versions of M3T-MR79 (for the 79xx series, 7700 family)

3.2 Description

If a set_flg or iset_flg service call is issued to handle an event flag for which tow or more tasks are waiting, and the interrupt handler that is invoked by an interrupt requested while the above service call is executed issues a set_flg service call to handle the above event flag, the waiting state of a task may not be released whereas it would normally be released by the set_flg or iset_flg service call issued at first.

3.3 Conditions

This problem may occur if the following conditions are all satisfied:

(1) Two or more tasks join a queue waiting for an event flag.

(2) A set_flg or iset_flg service call is issued to handle the event flag in (1), where the service call takes an argument the bit pattern of the event flag satisfying the waiting conditions of the tasks joining the queue.

(3) An interrupt is requested while set_flg or iset_flg in (2) is executed, and the invoked interrupt handler issues an iset_flg service call to perform the processing* for clearing the event flag in (1).

*The processing for clearing event flags is performed in the following cases:

(a) Issuing the iset_flg service call in (3) to handle an event flag of property TA_CLR releases both the waiting state of the task waiting for the event flag and the bit pattern of the event flag. This is the case only when M3T-MR30/4, M3T-MR308/4, or M3T-MR32R/4 used.

(b) Issuing the iset_flg service call in (3) releases both the waiting state of the task waiting for the event flag under the waiting condition with TWF_CLR and the bit pattern of the event flag. This is the case only when M3T-MR30, M3T-MR308, M3T-MR32R, and M3T-MR79 used.

(c) Issuing an iclr_flg or clr_flg service call as well as iset_flg in (3) changes the bit pattern of the event flag.

3.4 Workaround

Before and after the issuance of the set_flg or iset_flg service call described in Condition (2), disable and enable interrupts respectively. For examples, see Section 1.4

3.5 Schedule of Fixing the Problem

We plan to fix this problem in the next release of all the products except M3T-MR79, which remains unfixed.