Application Note, V1.0, Oct. 2006

AP08053

XC800 Family

RMAP and Paging in Interrupt Events

8bit

Microcontrollers

Never stop thinking

Edition 2006-10

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

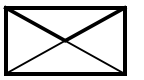
XC800 Family

Revision History: V1.0, 2006-10

Previous Version(s):
none

Page	Subjects (major changes since last revision)

We Listen to Your Comments
Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com



1 Overview

In the XC800 architecture, the Special Function Registers (SFRs) occupy the direct internal data memory space in the range 80_H to FF_H. However, the 128-SFR range is less than the total number of registers required and therefore address extension mechanisms are used to increase the number of addressable SFRs. The address extension mechanisms include:

- Mapping
- Paging

This document is intended to provide users of the Infineon XC800 family with guidelines on how to use the address extension mechanisms to improve code efficiency.

2 Address Extension by Mapping

The SFR can reside in two areas in the memory: the standard (non-mapped) SFR area and the mapped SFR area. Each area supports the same address range 80_H to FF_H bringing the number of addressable SFRs to 256. In order to access the mapped SFR area, the SFR bit in SYSCON0.RMAP must be set. Alternatively, the standard SFR area can be accessed by clearing bit RMAP.

SYSCON0
System Control Register 0 **Reset Value: 04_H**

7	6	5	4	3	2	1	0
	0		**IMODE**	**0**	**1**	**0**	**RMAP**
	r		rw	r	r	r	rw

The function of the shaded bit is not described here

Field	Bits	Type	Description
RMAP	0	rw	**Special Function Register Map Control** 0 Accessed to non-mapped(standard) special function register area. 1 Accessed to mapped special function register area.

In summary, accessing a mapped SFR will require the following steps:

1. Set RMAP
2. Access the SFR
3. Clear RMAP

Since every mapped SFR access can be interrupted by a higher priority interrupt, there might be a problem if an interrupt comes between step 1 and step 2. The code within the interrupt service routine will clear the RMAP bit if it needs to access the standard SFR area.

There are two methods to avoid the problem:

a) Block the interrupt globally before accessing the RMAP and enable globally after accessing RMAP or
b) Save the RMAP in every interrupt service routine, i.e. push RMAP on stack, modify RMAP accordingly and pop it back from stack.

Method (b) is preferred because disabling the interrupts globally can increase interrupt latency and might result in the loss of an interrupt event.

Examples for mapped peripherals:

WDT, UART1, T21, MDU, CORDIC, OCDS, Flash Routines (read and write)

Solution with method a:

```
refresh_WDT () {
EA=0;                   //block interrupts globally
SYSCON0 |= 0x01;        //set RMAP to access WDT register
WDTCON |= 0x02;         //refresh
SYSCON0 &=~0x01;        //clr RMAP
EA=1;                   //disable interrupts globally
}
```

Solution with method b:

```
refresh_WDT () {
SYSCON0 |= 0x01;        //set RMAP to access WDT register
WDTCON |= 0x02;         //refresh
SYSCON0 &=~0x01;        //clr RMAP
}

ISR_1 () {
_push_(SYSCON0);        //save RMAP status on stack
SYSCON0 &=~0x01;        //clr RMAP as an example for modification
...                     //code in interrupt service routine
_pop_(SYSCON0);         //restore RMAP status from stack
}
```

Note: Keil provides the following intrinsic functions for C51 in intrins.h to push and pop registers to and from the stack:

```
extern void          _push_    (unsigned char _sfr);
extern void          _pop_     (unsigned char _sfr);
```

3 Address Extension by Paging

The 256-SFR range is less than the total number of SFR needed by the on-chip peripherals. In order to meet this demand, some of these peripherals have a built-in local address extension mechanism, where additional address lines are added to decode the SFR of the peripheral kernel.

In order to access a register located in a different page, the bitfield PAGE bits in the page register can be programmed and the current page is left to access the new page. It should be noted that each peripheral that supports paging has it's own page register.

Accessing a page will require the following steps:

1. Set the page register Page bits
2. Access the module SFR

If an interrupt occurs between step 1 and step 2, and the interrupt must access a register located in another page, the current page setting must be saved, the new one programmed and finally, the old page restored after the interrupt service routine (ISR) is served.

The following example will illustrate the actions taken in an ISR.

1. Store RMAP
2. Set RMAP = 0 (all page registers reside in the standard SFR area)
3. Store current page setting
4. Program new page
5. Access module SFR
6. Other interrupt task
7. Restore old page setting
8. Restore RMAP

In all cases, the current page setting of the page registers must be saved before they are modified so that they can be restored to their original page when exited from the ISR. The most direct way to do this is to save the page registers to the stack. However, this method of saving the page registers to the stack is inefficient and it is especially significant with short ISR. It will not only increase the latency of the ISR but also makes use of the limited amount of stack space.

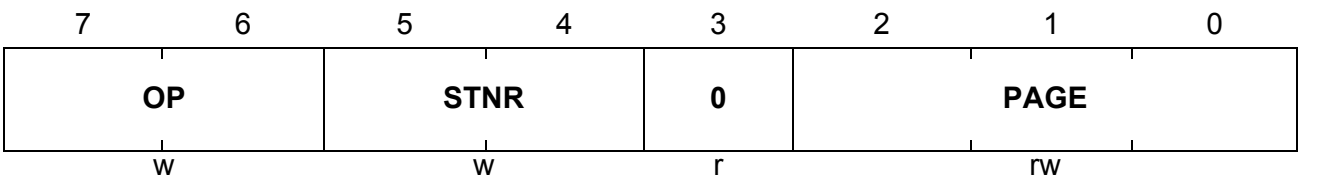
The XC800 architecture provides a more efficient mechanism to save and modify the current page setting without using the stack. This paging mechanism contains two

storage bits (4 storage containers) for the save and restore action. By indicating which storage bit field should be used in parallel to the new page value, a single write operation can save the current page setting and program a new page value. The above example is thus simplified by combining step 3 and step 4.

With this mechanism, a certain number of interrupt routines or other routines can do page changes without storing the formerly used page information. This will make the program simpler and faster. As a result, this mechanism will significantly improve the performance of a short ISR.

The page register has the following definition.

MOD_PAGE
Page Register for module MOD **Reset Value: 00_H**



Field	Bits	Type	Description
PAGE	[2:0]	rw	**Page Bits** When written, the value indicates the new page address. When read, the value indicates the currently active page.
STNR	[5:4]	w	**Storage Number** This number indicates which storage bit field is the target of the operation defined by bit OP. If OP = 10_B, the content of PAGE is saved in STx before being overwritten with the new value. If OP = 11_B, the content of PAGE is overwritten by the content of STx. The value written to the bit positions of PAGE is ignored. 00 ST0 is selected. 01 ST1 is selected. 10 ST2 is selected. 11 ST3 is selected.

Field	Bits	Type	Description
OP	[7:6]	w	**Operation** 0X Manual page mode. The value of STNR is ignored and PAGE is directly written. 10 New page programming with automatic page saving. The value written to the bit positions of PAGE is stored. In parallel, the former content of PAGE is saved in the storage bit field STx indicated by STNR. 11 Automatic restore page action. The value written to the bit positions PAGE is ignored and instead, PAGE is overwritten by the content of the storage bit field STx indicated by STNR.
0	3	r	**Reserved** Returns 0 if read; should be written with 0.

3.1 Using the Storage Containers

The paging mechanism provides a depth of four storage containers (ST0 - ST3) when there could be six levels of priorities. These levels (arranged according to the priority with 6 being the highest) are as follows:

1. Main
2. Interrupt level 0
3. Interrupt level 1
4. Interrupt level 2
5. Interrupt level 3
6. Non-maskable interrupt (NMI)

The main refers to routines that run prior to any interrupts and can be interrupted by any of the interrupts. The four interrupt levels (level 0 – 3) are four priority levels where each interrupt source can be programmed to. An interrupt that is currently being serviced can only be interrupted by a higher-priority interrupt, but not by another interrupt of the same or lower priority. Hence, an interrupt of the highest priority cannot be interrupted by any other interrupt request. In any case, the NMI always has the highest priority (above level 3) and its priority level cannot be programmed.

For any of the six levels mentioned above, the storage number should be unique at each level to avoid being overwritten by a different storage number when it is interrupted by a higher priority interrupt that is accessing the same module. Users must also ensure that the storage numbers within the ISRs are changed accordingly when the interrupt priority levels are changed.

Although there are six levels of priority as mentioned above, the main routine need not make use of the storage container mainly because it always has to open the correct page prior to accessing the module SFR and therefore it is not required to restore a previous page setting. This leaves us with five levels of interrupts and four storage containers. If all priority levels are used in an application, then not every interrupt level can have its own storage container. A workaround is to make use of the stack as the extended storage container. The ISR may also call functions that will modify the page registers. If these functions are shared by ISRs of different priority levels then these functions can be interrupted. It is therefore necessary to save and restore the page registers that will be modified in these functions. In such cases, the stack should be used as the storage container as the storage number of the page registers can be overwritten by a higher priority ISR calling the same function.

In summary:

- All the page registers modified in an ISR must be saved.
- The storage container should be unique at each interrupt level.
- The storage numbers within the ISRs must be changed accordingly when the interrupt priority levels are changed.
- No storage container is necessary for the main level.
- Stack can be used as the extended storage container.
- Page registers modified in functions called by the ISRs should use the stack as the storage container if the functions are shared among different priority level ISRs.

The example below will illustrate why it is necessary to use a different container for every interrupt level.

```
Main:
SCU page set as 1
Interrupt by level 0 ISR
Access ID register
…

ISR level 0:
```

```
Save current SCU page in storage 0 (ST0) and open SCU page 0 (SCU
page 1 in ST0)
Interrupt by level 1 ISR
Access IRCON1 register
Restore SCU page from ST0

ISR level 1:
Save current SCU page in storage 1 (ST1) and open SCU page 3 (SCU
page 0 in ST1)
Interrupt by level 2
Access IRCON3 register
Restore SCU page from ST1

ISR level 2:
Save current SCU page in storage 2 (ST2) and open SCU page 0 (SCU
page 3 in ST2)
Interrupt by level 3
Access IRCON2 register
Restore SCU page from ST2

ISR level 3:
Save current SCU page in storage 3 (ST3) and open SCU page 1 (SCU
page 0 in ST3)
Interrupt by NMI
Access PMCON0 register
Restore SCU page from ST3

NMI:
Save current SCU page in storage stack
Open SCU page 1
Access PLL_CON register
Restore SCU page from stack
```

Consider the case where NMI makes use of the storage container used in ISR level 1 (storage 1).

```
NMI:
Save current SCU page in storage 1 (ST1)and open SCU page 1 (SCU
page 1 saved in ST1)
Access PLL_CON
Restore SCU page in ST1
```

The pages will be restored correctly as the program returns from ISR at level 3 and level 2. However, the ISR at level 1 will restore SCU page 1 instead of SCU page 0 as the storage has been overwritten in the NMI routine. As a result, the ISR at level 0 will access the wrong module SFR. Although such scenarios are rare in a real application, caution must be exercised when handling the storage containers to avoid corruption of the page registers.

It may be appropriate for the interrupt at the lowest priority level to make use of the stack as the extended storage container since they are the least critical ones and therefore the latency may be more relaxed. It may also depend on other factors such as the number of page registers to save (stack space) and also the number of routines at that priority level (more routines mean more code).

3.2 Coding Examples

A page can be set active by writing directly to page register's page field (bits 0 - 2) with all other bits set to 0. The examples below show how SCU page 1 and PORT page 2 are set active.

```
SCU_PAGE = 0x01;
PORT_PAGE = 0x02;
```

The next example will illustrate how to save the current page of SCU_PAGE register into the storage bit field ST1 indicated by the STNR (bits 4 and 5) field and then restore it.

```
//SCU_PAGE was set to page 1
SCU_PAGE = 0x93 //open SCU page 3 and save SCU page 1 into ST1.
                //OP = 10, STNR = 01 (ST1), PAGE = 011
//… access module SFR in SCU page 3
SCU_PAGE = 0xD0 //restore page saved in ST1. OP = 11, STNR = 01
                //(ST1)
//SCU_PAGE restored to page 1
```

A macro is defined in main.h to open and save a page when code is generated from DAVE:

```
#define SFR_PAGE(pg, op) pg+op
```

Hence the above example can be written as:

```
//SCU_PAGE was set to page 1
SFR_PAGE(SCU_PAGE=3, 0x90);//open SCU page 3 and save SCU
                     //page 1 into ST1.
```

```
                        //OP = 10, STNR = 01 (ST1), PAGE = 011
//… access module SFR in SCU page 3
SFR_PAGE(SCU_PAGE=0, 0xD0);//restore page saved in ST1. OP = 11,
                        //STNR = 01 (ST1)
//SCU_PAGE restored to page 1
```

DAVE generated codes may save the current page to the storage every time a new page is written. At the main level, this is not necessary, however, it does not matter if you assign storage for it since the new page is always written prior to an access to the module SFR. It is only critical at ISR levels where the routines have to restore the page prior to the interrupt.

```
my_routine () {
//SCU_PAGE was set to page 0
SFR_PAGE(SCU_PAGE=1, 0xB0);//open SCU page 1 and save SCU page 0
                        //into ST3.
                        //OP = 10, STNR = 11 (ST3), PAGE = 001
… level 3 interrupt comes at this point!
//page is restored correctly although ST3 was corrupted in ISR_3()
//access module SFR on SCU page 1
PMCON0 = 0;
}


ISR_3 () {
_push_(SYSCON0);        //save RMAP status on stack
SYSCON0 &=~0x01;        //clr RMAP in order to access the standard
                        //memory where the SCU_PAGE registers
                        //reside.
SFR_PAGE(SCU_PAGE=3, 0xB0); //open SCU page 3 and save current
                        //SCU page into ST3.
                        //OP = 10, STNR = 11 (ST3), PAGE = 011
                        //overwrites ST3 in my_routine ()
//access module SFR on SCU page 3
IRCON3 = 0;
… other code in interrupt service routine
SFR_PAGE(SCU_PAGE=3, 0xF0);//restore SCU page from ST3
                        //OP = 11, STNR = 11 (ST3), PAGE =
                        //ignore
//SCU page restored to page 1
_pop_(SYSCON0);         //restore RMAP status from stack
}
```

Saving a page register to the stack will be very direct. Using the above example,

```
my_routine () {
//SCU_PAGE was set to page 0
SFR_PAGE(SCU_PAGE=1, 0xB0);//open SCU page 1 and save SCU page 0
                     //into ST3.
                     //OP = 10, STNR = 01 (ST1), PAGE = 011
… level 3 interrupt comes at this point!
//access module SFR of SCU page 1
PMCON0 = 0;
}


ISR_3 () {
_push_(SYSCON0);     //save RMAP status on stack
SYSCON0 &=~0x01;     //clr RMAP in order to access the standard
                     //memory where the SCU_PAGE registers
                     //reside.
_push_(SCU_PAGE);    //save SCU_PAGE in stack
SFR_PAGE(SCU_PAGE=3, 0x00); //Open SCU page 3 without saving the
                     //current page.
//access module SFR on SCU page 3
IRCON3 =0
… other code in interrupt service routine
_pop_(SCU_PAGE);     //SCU page restored to page 1
_pop_(SYSCON0);      //restore RMAP status from stack
}
```

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG