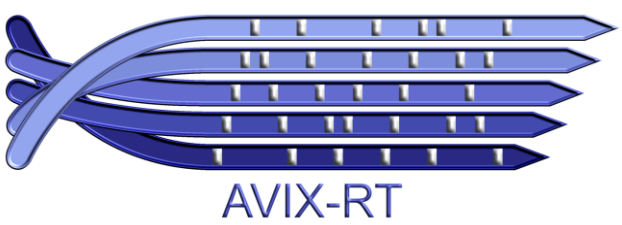
AVIX Real Time Operating System

AVIX-RT Technical Paper

Thread Safe use of Special Function Registers



AVIX-RT
Maïsveld 84
5236 VC 's-Hertogenbosch
The Netherlands

phone +31(0)6 15 28 51 77
e-mail info@avix-rt.com
www www.avix-rt.com

Table of contents

1 Introduction

This document provides a description of an often underestimated problem in microcontroller programming, a problem many developers are not even aware of, a problem potentially leading to spurious errors which are easy to introduce and very hard to solve. It is strongly advised to at least read this chapter to learn what this problem is

Software running on a microcontroller almost always is composed of multiple active entities. In its most basic form these are the main program loop and one or more Interrupt Service Routines (ISR's). When using an RTOS, the active entities are the threads running under control of the RTOS and one or more ISR's. A characteristic of an active entity is that it pre-empts another active entity at an unpredictable moment. You just cannot predict what the microcontroller is doing when an interrupt occurs, that is the essence of an interrupt. When using an RTOS, in principle you don't know where one thread is interrupted in order for another thread or ISR to be activated.

Microcontrollers offer I/O capabilities through Special Function Registers (SFR's). Most SFR's contain multiple bit-fields, each with a different function. It is very likely that the different bit-fields offered by the SFR's are used from different active entities.

This can lead to a very serious problem, the infamous Read-Modify-Write problem. A static analysis of the program can reveal one SFR bit-field to be used by one active entity and another bit-field *in the same SFR* by another active entity and everything seems fine. When running the program however it appears that these actions are not independent and manipulation of one bit-field disturbs the manipulation of the other bit-field. The program is not working as it should be. This category of errors often is very hard to find since the error occurs at seemingly random moments. The underlying cause of this problem is explained in more detail in section 2 of this document.

Another characteristic of this category of errors is that many developers are not even aware they exist. Software is tested and everything seems to work fine but still the error can be there in order to show up when it causes the most damage. Another possibility is the problem exists but because of compiler optimizations it does not occur. A new version of the compiler may contain a different optimization algorithm leading to the error showing up in a maintenance phase.

In order to prevent this category of errors, manipulation of individual SFR bit-fields must be atomic meaning the result of the operation is not disturbed by interrupts, regardless when the interrupt occurs. There are many ways to accomplish this. The most basic one is to disable interrupts when the operation is being performed. For the PIC24/33 platform manipulating single-bit SFR fields is atomic by using the bclr (bit clear) or bset (bit set) instructions. This does however not work for multiple-bit SFR fields. The PIC32 platform offers special registers allowing atomic bit field manipulation. Problem is these mechanisms are not uniform and do not cover all situations.

As part of the AVIX RTOS, a number of utility macros are provided that fully prevents this problem. These macros are used from a C program whenever the program accesses SFR fields, be it a single-bit or a multiple-bit field, and be it a read or a write operation.

The remainder of this document contains a more detailed description of the problem and the AVIX supplied macros that can be used to prevent this problem.

2 The problem

As explained in the introduction, the problem addressed in this document is known as the Read-Modify-Write problem. This is first illustrated with a generic example which will be detailed to the SFR situation later. The Read-Modify-Write problem is always related to multiple active entities manipulating the same memory location, an SFR in this case. Suppose a program exists where an ISR increments a counter variable and the main loop waits for this flag to become unequal zero, perform some action and subsequently decrements the counter. In general, updating the counter variable comes down to reading the content of the variable into a processor internal register, perform the desired operation on this register and write the resulting value back to the counter variable. So in pseudo code this looks like:

```
ISR                                     Main loop
…;                                      …;
read variable to register;              read variable to register
increment register content;             decrement register content
write register back to variable;        write register back to variable
…;                                      …;
```

Let's assume now the content of the variable is 1 and the main loop has just read this value into the processor internal register when the interrupt occurs. The ISR runs to completion and will increment the value of the variable leaving it with value 2. After the ISR has executed, the main loop will continue where it was interrupted. Since the main loop already read the variable content being 1, it will continue there, decrement the register content to 0 and write this back to the variable. As a result the increment performed by the ISR is lost!

The same problem can happen when an SFR contains two bit fields where one field is manipulated by the ISR and the other by the main loop. This is illustrated by the use of the IC1CON register present in the 32MX360F512L controller. Bit 2..0 is the ICM field, specifying the input capture mode. Bit 6..5 is the ICI field, specifying on which capture event an interrupt occurs. Suppose now the ICM field is manipulated by the main loop and the ICI field by an ISR. When just using the IC1CON register from a C program, the compiler will translate both register manipulations to a Read-Modify-Write sequence. The current value of the entire SFR is read to a processor internal register, in the processor internal register, the applicable bits are set to the desired value and finally the content of the processor internal register is written back to the memory location where SFR IC1CON lives.

Based on moment the interrupt occurs, the same problem as illustrated above can happen, effectively destroying the most recent manipulation of the ICI field by the ISR.

3 The solution

As said in the Introduction, there are many solutions to this problem most having a lot of drawbacks. The most brute force approach is to disable interrupts when an atomic manipulation is required. This is however easy to forget, more code needs to be written and actually is not what you want in a real time system. Other solutions are platform specific introducing code differences at application level between the PIC24/30/33 platform and the PIC32 platform. The solution chosen by AVIX-RT is based on the fact that both hardware platforms do offer either instructions or special registers allowing basic assembly instructions to be performed atomically. By exploiting these capabilities, a set of macros is created allowing atomic manipulation of SFR bit fields where the application programmer uses the same macro syntax regardless the platform used. These macros hide all underlying differences leading to a portable solution. In the next sections a description is given of the mechanism, its design goals and how to use it.

3.1 Design goals

The following design goals were set for the mechanism presented in this document:

- **Efficiency**[1]: Code generated by the mechanism must be comparable regarding efficiency to code generated when directly accessing SFR's. This goal is fully met and code generated by this mechanism is often even more efficient than code generated by direct SFR access.

- **Single definition**: When defining a port for a specific purpose, an application typically contains a section containing a number of #defines for this purpose. Design goal for this mechanism was that a definition for a bit field in an SFR would only have to be made once where this single definition can be used for writing or reading the bit field.

- **Portability**: Applying the mechanism must be possible over all controllers targeted by the different AVIX distributions without any change to the applications source code.

3.2 How to use the mechanism

For all I/O ports and devices offered by PIC24/30/33 and PIC32 definitions are provided allowing a C program to use them. Most I/O pins can be used in more than one manner. The most basic usage is as a direct digital I/O pin. This usage is supported by the PORT, TRIS and LAT register definitions. Using these definitions a program is in control how the pins are used. Alternatively a higher level device can be used like SPI, UART, I2C etc. These devices use the same hardware pins. Usage of a device precludes direct I/O pin manipulation of those pins used by that device. The related register and bit field definition is device specific.

The here described AVIX SFR mechanism uses Microchip provided I/O port/device definitions. Just like pin manipulation coming in two distinct ways, either direct or through implicit usage by one of the on-chip devices, so comes the AVIX SFR mechanism in the form of two distinct macros. These macros are `avixSFR` and `avixSFR_Range`. The first macro is used for manipulating a SFR bit field based on the bit field definition from a Microchip provided header file. The second macro is used for manipulating a multiple bit field in an SFR where the bit field is user defined. These macros can be used to either Write, Read or Invert an SFR bit field.

Usage is illustrated with a sample again based on the IC1CON register. The Microchip provided header files contain the following definition:

[1] The here presented mechanism does require code to be compiled with optimization level1 (-O1) or higher in which case the mechanism generates code often even more efficient than direct SFR manipulation. Compiling without optimization does generate more code than the basic solution and is strongly discouraged.

```
typedef union {
  struct {
    unsigned ICM0:1;
    unsigned ICM1:1;
    unsigned ICM2:1;
    unsigned ICBNE:1;
    unsigned ICOV:1;
    unsigned ICI0:1;
    unsigned ICI1:1;
    unsigned ICTMR:1;
    unsigned C32:1;
    unsigned FEDGE:1;
    unsigned :3;
    unsigned SIDL:1;
    unsigned FRZ:1;
    unsigned ON:1;
  };
  struct {
    unsigned ICM:3;
    unsigned :2;
    unsigned ICI:2;
    unsigned :6;
    unsigned ICSIDL:1;
  };
  struct {
    unsigned w:32;
  };
} __IC1CONbits_t;
```

Regular usage of this definition leads to the following code when for instance manipulating the ICM and ICI fields:

```
IC1CONbits.ICM = 3;
IC1CONbits.ICI = 2;
```

The above operations are not atomic and when the ICM field is manipulated from another thread than the ICI field (or one field is manipulated from main code and the other from an ISR) potentially leads to the aforementioned problem.

Using the AVIX supplied macro; these operations are coded as follows:

```
avixSFR(IC1CON, ICM, WRITE_BF, 3);
avixSFR(IC1CON, ICI, WRITE_BF, 2);
```

Functionally this results in the same code but as a benefit manipulating the ICM field from one thread and the ICI field from another (or one from main code and the other from an ISR), these manipulations do not influence each other. There is no need to disable interrupts and as long as a single field is manipulated from a single active entity, the field manipulation will be correct and not influenced by other active entities manipulating other fields of the same register.

It is advised not to code the above construct directly in your source code but make definitions based on this syntax and use these instead. The advantage of this approach is that the actual SFR selection (IC1CON in this case) is present in one place only. Doing so the above example could look like:

```
/* Definitions
*/
#define MYICM_DEF(v) avixSFR(IC1CON, ICM, WRITE_BF, (v))
#define MYICI_DEF(v) avixSFR(IC1CON, ICI, WRITE_BF, (v))

/* Usage
*/
MYICM(3);
MYICI(2);
```

Another reason for making such definitions is when a bit field must both be written and read. By using a central definition, again the actual SFR and bit field is only present in one place in your application, minimizing the risk on making errors when changes have to be made to those definitions. The definition for a bit field that must be written and read looks as follows:

```
/* Definitions
*/
#define MYICM_DEF(a,…) avixSFR(IC1CON, ICM, (a), __VA_ARGS__)
#define MYICI_DEF(a,…) avixSFR(IC1CON, ICI, (a), __VA_ARGS__)

/* Write Usage
*/
MYICM(WRITE_BF, 3);
MYICI(WRITE_BF, 2);

/* Read Usage
*/
int valICM = MYICM(READ_BF);
int valICI = MYICI(READ_BF);
```

The AVIX macros use variable arguments. Reason is when setting a bit field, two arguments are needed (the WRITE_BF command and the value) and when reading a bit field only a command is sufficient (READ_BF). The syntax shown above is standard C. The ellipsis (…) identify that here zero or more arguments are allowed and the __VA_ARGS__ in the macro expansion is a placeholder for those arguments. When the macro is parsed, __VA_ARGS__ is replaced with all arguments passed in place of the ellipsis (…).

Strictly spoken, reading a bit field does not need to be based on the AVIX macros. When reading a bit field there is no Read-Modify-Write problem. Reading will always return the current value without the risk the SFR is corrupted by concurrent actions. The advantage of using the AVIX macros for read operations also is that your application does contain a single definition of the actual SFR and bit field greatly reducing the risk of making errors when the register and/or bit field has to be changed.

Finally, avixSFR supports inverting the value of the bit field by using the INVERT_BF command. This will perform an atomic inversion of the bit field, an operation most suitable for single bit bit fields.

The second macro supplied is `avixSFR_Range`. This macro is used when accessing the basic ports where the individual bits have an application defined meaning. Again this is illustrated by an example. Suppose an LCD is connected to the PIC where data is transferred on port E bit 0 up to bit 3. avixSFR_Range allows this range of bits to be specified. In order to control the LCD, this would imply a definition for the TRIS register and for the LAT register. Assuming the applicable bits only need to be written, the definitions and usage could be as follows:

```
/* Definitions
*/
#define MYLCD_TRIS(v) avixSFR_Range(TRISE, 0, 3, WRITE_BF, (v))
#define MYLCD_LAT(v)  avixSFR_Range(LATE,  0, 3, WRITE_BF, (v))

/* Usage
*/
MYLCD_TRIS(0);      /* Set port E bit 0..3 as output        */
        …;
MYLCD_LAT(data);    /* Write a nibble to port E bit 0..3    */
```

Again, when using these macros from a single active entity, the bits are guaranteed to receive the desired values without influencing the value of other bits in the same port. Likewise, when other active entities manipulate other bits in the same port, these operations are guaranteed not to corrupt the operations on bit 0..3. As long as a single active entity writes a single bit field, the operation is guaranteed to be atomic.

Apart from the field being defined through a numeric range, this macro has the same functionality and usage as `avixSFR`.

3.3 API specification

The functionality described in this document is present in file AVIXAtomicSFR.h which is installed from the AVIX installation program when selecting to install additional utilities.

The following macros are offered:

```
avixSFR(r, f, c, …);
```

Description: Write, read or invert a bit-field in a special function registers based on the controller specific SFR structure definitions.

Arguments: The following arguments are passed to this function:

- **r**: Name of the Special Function Register as specified in the applicable Microchip header file.
- **f**: Name of the field in the Special Function Register as specified by argument r.
- **c**: Command code, allowed values are WRITE_BF, READ_BF or INVERT_BF
- …: In case argument c has value WRITE_BF, the value that has to be written in the designated field. Note that when writing a value to the bit field, the least significant bits of the value are used where the number of bits equals the number of bits in the bit field being written. Remaining bits are ignored without warning. So writing the value 5 to a bit field two bits in length, effectively writes value 1 to those bit field, being the lower two bits of value 5.

Return value: In case argument c has value READ_BF only, the value of the applicable bit field.

When specifying a command value different than WRITE_BF, READ_BF or INVERT_BF, the macro generates a compile time error of the form: `_avixSFR_IllegalCommandValue_0' undeclared (first use in this function)

```
avixSFR_Range(r, l, u, c, …);
```

Description: Write, read or invert a bit-field in a special function registers where the bit field is identified as a numeric range.

Arguments: The following arguments are passed to this function:

- **r**: Name of the Special Function Register as specified in the applicable Microchip header file.
- **l**: Lower bit number of bit field
- **u**: Upper bit number of bit field.
- **c**: Command code, allowed values are WRITE_BF, READ_BF or INVERT_BF
- …: In case argument c has value WRITE_BF, the value that has to be written in the designated field. Note that when writing a value to the bit field, the least significant bits of the value are used where the number of bits equals the number of bits in the bit field being written. Remaining bits are ignored without warning. So writing the value 5 to a bit field two bits in length, effectively writes value 1 to those bit field, being the lower two bits of value 5.

Return value: In case argument c has value READ_BF only, the value of the applicable bit field.

When specifying a command value different than WRITE_BF, READ_BF or INVERT_BF, the macro generates a compile time error of the form: `_avixSFR_IllegalCommandValue_0' undeclared (first use in this function)

4 How efficient is the mechanism?

One of the design goals was that code generated by the mechanism must be comparable regarding efficiency to code generated when directly accessing SFR's. When using the mechanism with optimization level 1 or higher, this goal is met as is illustrated in this chapter. Using no optimization the mechanism does generate more code than the basic mechanism therefore when using this mechanism, it is strongly advised to compile your application using optimization level 1 (-O1) or higher. Still, when using no optimization the macro's work as specified from a functional point of view.

This chapter presents code efficiency benchmarks for the following situations:

Set single-bit field with constant value
Set single-bit field with variable value
Set multi-bit field with constant value
Set multi-bit field with variable value

All figures are presented for both the 16 and the 32 bit version. As shown in the tables on the following pages, the AVIX-16 bit mechanism performs equal or better than the basic mechanism under all circumstances. The AVIX-32 mechanism performs better when setting a single bit field to a constant value and equal or slightly worse for the other situations.

▶ *Keep in mind that alternative solutions like disabling interrupts of using a mutex add more code. This extra code is not present in this comparison. For this reason it is correct to state the AVIX mechanism is more efficient under all circumstances.*

▶ *Furthermore analysis of production code revealed that the majority of SFR operations are single bit with constant values. Therefore applying the mechanism has a positive influence on overall code size and performance.*

16 BIT VERSION[1]		
WRITE SINGLE-BIT FIELD WITH CONSTANT VALUE		
`LATAbits.LATA6 = 0;`	`bclr.b 0x02c4,#6`	Both the basic and the AVIX mechanism have the same performance.
`avixSFR(LATA,LATA6,WRITE_BF,0)`	`bclr.b 0x02c4,#6`	
WRITE SINGLE-BIT FIELD WITH VARIABLE VALUE		

`LATAbits.LATA6 = var;`	`mov.w var,w1`
	`and.b w1,#1,w1`
	`sl w1, #6,w1`
	`mov.b 0x02c4,w0`
	`bclr w0,#6`
	`ior.b w0,w1,w0`
	`mov.b w0,0x02c4`
		The basic mechanism takes 7 cycles. The AVIX mechanism takes 4 cycles when clearing a bit and 5 cycles when setting a bit.

`avixSFR(LATA,LATA6,WRITE_BF,var);`	`btst.b var,#0`
	`bra z, 0x003eb0`
	`bset.b 0x02c4,#6`
	`bra 0x003eb2`
	`bclr.b 0x02c4,#6`	

WRITE MULTI-BIT FIELD WITH CONSTANT VALUE		

`IC1CONbits.ICI = 1;`	`mov.b 0x0142,w0`
	`mov.b #0x9f,w1`
	`and.b w0,w1,w0`
	`bset w0,#5`
	`mov.b w0,0x0142`
		The basic mechanism takes 5 cycles. The AVIX mechanism takes 4 cycles.

`avixSFR(IC1CON,ICI,WRITE_BF,1);`	`mov.w 0x0142,w0`
	`btg w0,#5`
	`and.w #0x60,w0`
	`xor.w 0x0142`	

WRITE MULTI-BIT FIELD WITH VARIABLE VALUE		

`IC1CONbits.ICI = var;`	`mov.w var,w1`
	`and.b w1,#3,w1`
	`sl w1,#5,w1`
	`mov.b 0x0142,w0`
	`mov.b #0x9f,w2`
	`and.b w0,w2,w0`
	`ior.b w0,w1,w0`
	`mov.b w0,0x0142`
		The basic mechanism takes 8 cycles, the AVIX mechanism takes 7 cycles.

`avixSFR(IC1CON,ICI,WRITE_BF,var);`	`mov.w var,w0`
	`and.w w0,#3,w0`
	`sl w0,#5,w0`
	`mov.w 0x0142,w1`
	`xor.w w0,w1,w0`
	`and.w #0x60,w0`
	`xor.w 0x0142`	

[1] Figures based on C30 compiler version 3.11b with –O1 optimization

32 BIT VERSION[1]		
WRITE SINGLE-BIT FIELD WITH CONSTANT VALUE		

`LATAbits.LATA6 = 0;`	`lui a0,0xbf88`
	`lbu v0,24608(a0)`
	`addiu v1,zero,-65`
	`and v0,v0,v1`
	`sb v0,24608(a0)`
		The basic mechanism takes 5 instructions. The AVIX mechanism takes 3 instructions.

`avixSFR(LATA,LATA6,WRITE_BF,0)`	`addiu v1,zero,64`
	`lui v0,0xbf88`
	`sw v1,24612(v0)`	

WRITE SINGLE-BIT FIELD WITH VARIABLE VALUE		

`LATAbits.LATA6 = var;`	`lui a0,0xbf88`
	`lw v0,24608(a0)`
	`lw v1,-32168(gp)`
	`ins v0,v1,6,1`
	`sw v0,24608(a0)`
		The basic mechanism takes 5 instructions. The AVIX mechanism takes 5 instructions when clearing a bit and 7 instructions when setting a bit.

`avixSFR(LATA,LATA6,WRITE_BF,var);`	`lw v0,-32168(gp)`
	`andi v0,v0,0x1`
	`beq v0,zero,0x9d016530`
	`addiu v1,zero,64`
	`lui v0,0xbf88`
	`beq zero,zero,0x9d016538`
	`sw v1,24616(v0)`
	`lui v0,0xbf88`
	`sw v1,24612(v0)`	

WRITE MULTI-BIT FIELD WITH CONSTANT VALUE		

`IC1CONbits.ICI = 1;`	`lui a0,0xbf80`
	`lw v0,8192(a0)`
	`addiu v1,zero,1`
	`ins v0,v1,5,2`
	`sw v0,8192(a0)`
		The basic mechanism takes 5 instructions. The AVIX mechanism takes 6 instructions.

`avixSFR(IC1CON,ICI,WRITE_BF,1);`	`lui v0,0xbf80`
	`lw v0,8192(v0)`
	`xori v0,v0,0x20`
	`andi v0,v0,0x60`
	`lui v1,0xbf80`
	`sw v0,8204(v1)`	

WRITE MULTI-BIT FIELD WITH VARIABLE VALUE		

`IC1CONbits.ICI = var;`	`lui a0,0xbf80`
	`lw v0,8192(a0)`
	`lw v1,-32168(gp)`
	`ins v0,v1,5,2`
	`sw v0,8192(a0)`
		The basic mechanism takes 5 instructions. The AVIX mechanism takes 8 instructions.

`avixSFR(IC1CON,ICI,WRITE_BF,var);`	`lw v0,-32168(gp)`
	`sll v0,v0,0x5`
	`lui v1,0xbf80`
	`lw v1,8192(v1)`
	`xor v0,v0,v1`
	`andi v0,v0,0x60`
	`lui v1,0xbf80`
	`sw v0,8204(v1)`	

[1] Figures based on C32 compiler version 1.04 with –O1 optimization