WRITING DEVICE HEADER FILES

Document revision:	PA3
Date:	28 June 2005
Author:	Sara Skrtic

Revision log:

Revision	Date	Author	Modification
PA1	2005-05-06	Sara Skrtic	Initial preliminary revision
PA2	2005-06-03	Sara Skrtic	Interrupt values – added solution with conflicting bit names / interrupt names
PA3	2005-06-28	Sara Skrtic	Added MISRA C control

Useful documents

There are some useful documents to help you write header files.

- IAR code standards (CppGuide.html)
- IAR header file template (EWARM_HeaderTemplate.doc)

Where to start

A few pointers:

1. Find the relevant and up-to-date device user guide (can usually be downloaded from the chip manufacturer's web site).
2. Check if IAR Systems already supports more devices from the same family. You can often re-use parts from other header files belonging to the same chip family.

What is included in a header file?

A header file can be said to consist of 6 sections:

1. **A header** with the following information:
   - Which IAR Compiler and Assembler that the header file designed for
   - that the header file is used with ARM IAR C/C++ Compiler and Assembler
   - IAR Systems copyright information and the header file creation year
   - File revision: $ Revision $
2. **Protection against multiple inclusions of the same header file**
   - o by defining and testing on __iochipname_h

     ```
     #ifndef __IOCHIPNAME_H
     #define __IOCHIPNAME_H
     ```

   - o check if IAR compiler

     ```
     #if (((__TID__ >> 8) & 0x7F) != 0x4F)      /* 0x4F = 79 dec */
     #error This file should only be compiled using the ARM IAR compiler and assembler
     #endif
     ```

   - o by including the macro definition file (io_macros.h)

     ```
     #include "io_macros.h"
     ```

3. **Guard against MISRA C errors and Special Function Registers**.
   - o Guard against MISRA C errors with a pragma. This is necessary since the declaration of IO registers will otherwise cause MISRA C rule violation errors. The guard applies to this file only.

     ```
     #ifndef _SYSTEM_BUILD
          #pragma system_include
     #endif
     ```

   - o The bit structs containing the bit names of the registers.
4. **Common declarations** of the register groups.
5. **Assembler-specific declarations** (optional).
6. **Various symbol definitions**. These are most often related to interrupt values/numbers (optional).

IAR standards for header files

- **The header file name should be io*chipname*.h**

  Example:

  The header file for the chip *c1234* is called ioc1234.h

- **Registers are defined using the __IO_REG*XX* macros defined in io_macros.h.**

  - There are two formats used, one for registers without specific bit names and one for registers with specific bit names.

    Without bit names:

    ```
    __IO_REGXX(REG_NAME, REG_ADDR, INFO_ACCESS_TYPE);
    ```

    With bit names:

    ```
    __IO_REGXX_BIT(REG_NAME, REG_ADDR, INFO_ACCESS_TYPE, BIT_STRUCT_NAME);
    ```

    __IO_REGXX_

Name	Description
XX	The register width in bits
REG_NAME	The name of the register. Always use the name and format found in the device user guide.
REG_ADDR	The address of the register.
INFO_ACCESS_TYPE	The way the information is accessed. Always start with a double underscore. __READ __WRITE __READ_WRITE
BIT_STRUCT_NAME	The structure in which you name and order the bits of the register. See the section **The bit struct order** for further information

    Example without bit names:

    register REG_NAME_A is a 32-bit read-only register located at address 0x10000004 with no bit names defined.

    ```
    __IO_REG32(REG_NAME_A, 0x10000004, __READ);
    ```

    Example with bit names:

    register REG_NAME_B is a 16-bit read/write register located at address 0x10000008 with bit names defined.

    ```
    __IO_REG16_BIT(REG_NAME_B, 0x10000008, __READ_WRITE, __regnameb_bits);
    ```

  - To use the file with the assembler, there cannot be any space between __IO_REGXX and the left parentheses in a register definition.

    Example:

    ```
    __IO_REG16(    HDLC_TFBC,   0xC000E040, __READ_WRITE);   //Correct
    __IO_REG16    (HDLC_RFBC,   0xC000E044, __READ);       //Wrong
    ```

- **The correct bit struct order must be used.** When writing the bit struct definition, always start with bit 0 and end with the highest numbered bit. The total number of bits declared must be equal to the size of the bit field used (that is, in a struct using the __REG*XX* macro the total number of bits is *XX*). This requires that all reserved/unused bits are declared as well. Do this by not naming the bits - leave an empty space. Always start the name of the struct with a double underscore "__", the name of the register in lowercase letters and end with _bits.

  The format used:

  ```
  typedef struct {
        __REGXX    BITNAME1   : #_OF_BITS;
        __REGXX    BITNAME2   : #_OF_BITS;
  } __registername_bits;
  ```

  XX is the total length of the register in bits (8, 16 or 32).

BITNAME is the name of the bit(s). Always follow the name and format used in the device user guide.

#_OF_BITS is the number of bits used by *BITNAME*.

Example:

Device user guide:

7	6	5	4	3	2	1	0
Src	reserved		Oiop	reserved	Addr	reserved	

Header file:

```
typedef struct {
     __REG8       : 2;
     __REG8  Addr      : 1;
     __REG8            : 1;
     __REG8  Oiop      : 1;
     __REG8            : 2;
     __REG8  Src       : 1;
} __registername_bits;
```

- **The name and format (upper lowercase) of bits and registers must follow the device user guide**. Sometimes bits are referred to with descriptions instead of names. If that is the case, check to see if IAR Systems supports another chip from the same family. Chances are that the register and bit(s) have been named already. Otherwise, name the bit(s).

Example:

Device user guide:

Register X	**Bit**	**Description**
Interrupt Source	[7:4]	0000 source1 0010 source2 1001 source3
Reserved	[3:1]	Must be 0
Interrupt Enable	[0]	1 Enable 0 Disable

If the bits are named in another header file for a device in the same family – use that name, otherwise name it yourself (suggestions: IntSrc, IntEn)

- **Bit names cannot start with a number**. If the device user guide has a bit name that begins with a number, use a single underscore “_” before the number when you write the bit struct definition.

Example:

Device user guide:

7	6	5	4	3	2	1	0
Src			3ipn	reserved	Addr	reserved	

Header file:

```
typedef struct {
     __REG8       : 2;
     __REG8  Addr      : 1;
     __REG8            : 1;
     __REG8  _3ipn     : 1;
     __REG8  Src       : 3;
} __registername_bits;
```

- **Bit structs and register groups should be commented**. Before every bit struct – make a comment explaining which register it belongs to. Before every group of registers – make a comment explaining which group it is.

  Example:

```
/* DMA - Status Register */
typedef struct {
  __REG32  xx  :20;
  __REG32  y    :12;
} __dmastat_bits;

/*********************************************************************
**
**    DMA
**
*********************************************************************/
__IO_REG32_BIT(DMASTAT, 0x48000000, __READ_WRITE, __dmastat_bits);
__IO_REG32(DMASRC,    0x48000004, __READ_WRITE);
```

- **Place groups of registers and registers in the correct order.** Often in the device user guide there is a table listing all special function registers. In the device header file, place groups of registers in the same order as in that table. Within the groups try to place the registers in address order.

- **Interrupt values.** In the device user guide, there might be interrupt values/numbers that need to be defined. The instructions for doing this differ significantly from user guide to user guide. Some only describe the interrupts without naming them, while others specifically give names.
  - If there are names, follow the guide. If the names conflict with bit names, add “INO_” in front of the name.
  - If there are no specific names, you have to use your imagination. (There is an effort going on to make a list of standard interrupt names.)

Solving problems

- **Problem: Which address should be used for an 8- or 16-bit register on a 32-bit device?**

Base address: 0x0000_0000	*Bit*																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Byte #																															
Big-endian	0								1								2								3							
Little-endian	3								2								1								0							

Solution: This depends on the endian the device is using. Sometimes the device user guide is giving a specific address even for 8- and 16-bit registers; otherwise use the base address and the byte number as described above.

An 8-bit, one byte, read from a 32-bit register on a big-endian device would return bit [31:24]

An 8-bit, one byte, read from a 32-bit register on a little-endian device would return bit [7:0]

Normally a device supports either little- or big-endian byte order, but some support both. If it is a full size register there is no problem (that is, a 32-bit register on a 32-bit device or a 16-bit register on a 16-bit device). Otherwise, be careful with the addresses. There is a predefined symbol called __LITTLE_ENDIAN__ that will expand to one (1) when the code is compiled with the little-endian byte order format, and zero (0) if the code is compiled with the big-endian format. Use #if … #else ... #endif.

Example:

```
__IO_REG32(REG_NAME1, 0x40000000, __READ);  //big- or little-endian
#if __LITTLE _ENDIAN__
__IO_REG8(REG_NAME2, 0x40000004, __READ_WRITE);
#else
__IO_REG8(REG_NAME2, 0x40000007, __READ_WRITE);
#endif     /*__LITTLE_ENDIAN__*/
```

- **Problem: Two registers share the same address**

  Solution: If two registers share the same address, either in a situation where bit names are not defined, or where the bit names are also identical, use a #define

  Example 1:

  Register REG_NAME_A is a 32-bit read only register located at address 0x10000008 with no bit names defined.

  Register REG_NAME_B is a 32-bit read/write register located at the same address with no bit names defined. Note in the following example that although REG_NAME_A is a read-only register it must be read/write for REG_NAME_B to work.

```
__IO_REG32(REG_NAME_A, 0x10000008, __READ_WRITE);
#define REG_NAME_B       REG_NAME_A
```

  Example 2:

  REG_C is an 8-bit read-only register located at address 0x10000014
  Bit [0] is called EN and the rest of the bits [7:1] are reserved
  REG_D is a write-only register that otherwise looks exactly the same as REG_C.

```
/* C-compiler specific declarations *********************************/
typedef struct {
   __REG8 EN     : 1;
   __REG8        : 7;
} __regn_bits;

/* Declarations common to compiler and assembler *********************/
__IO_REG8_BIT( REG_C, 0x10000014, __READ_WRITE, __regn_bits);
#define REG_D         REG_C
#define REG_D_bit     REG_C_bit
```

- **Problem: Several registers using different bit names share the same address**

  Solution: Use a union of structs and #define. Observe that
  1. bit names only can be used once in the same union.
  2. before the struct you should make a // comment with the register name. If the register name contains a wildcard character for numbers it can only be x or y and only in lowercase (for instance REGx.)

  Example:

  REG_E is an 8-bit read-only register located at address 0x10000014
  Bit [0] is called EN and the rest of the bits [7:1] are reserved

  REG_F8 and REG_F9 are read/write registers located at the same address
  Bit [0] is called EN
  Bits [4:1] are reserved
  Bits [7:5] are called ST

```
/* C-compiler specific declarations *********************************/
typedef union {
   //REG_E
   struct {
         __REG8 EN  : 1;
         __REG8     : 7;
      };
      //REG_Fx
      struct {
         __REG8          : 1;
         __REG8          : 4;
         __REG8  ST      : 3;
      };
} __regx_bits;

/* Declarations common to compiler and assembler *********************/
__IO_REG8_BIT(REG_E, 0x10000014, __READ_WRITE, __regx_bits);
#define REG_F8         REG_E
#define REG_F8_bit     REG_E_bit
#define REG_F9         REG_E
#define REG_F9_bit     REG_E_bit
```

- **Problem: One register has different bit names**

  Solution: Sometimes there can be registers that name bits differently depending on different objectives (source, read or write et cetera). Use a union of structs, see the example with several registers using different bit names sharing the same address, with the two // comments naming the same register name.

- **Problem: I have wide registers on a 32-bit device**

  Solution: Wide registers, for example 48- or 64-bit registers, have to be divided into two parts called High and Low.

  Example:

  A register REG_G is a 64-bit read/write register located at address 0x10000000 on a little-endian device.
  ```
  __IO_REG32(REG_G_Low,  0x10000000, __READ_WRITE);
  __IO_REG32(REG_G_High, 0x10000004, __READ_WRITE);
  ```

- **Problem: There are two or more devices with the same registers**

  Solution: This is a nice “problem” to have. If there are two or more devices that have the same register setup, make a “dummy” header file in which you include the first header file.

  Example:

  Two devices, x1000 and x1001 have the same registers. Create the header file iox1000.h in which you place all the registers and structs. Then make an empty file called iox1001.h in which you put only the header (see **What is included in a header file**) and a #include “iox1000.h”.

Ending the header file

Remember to always end your file with an end-of-line character (EOL).