PRECISION32™ CMSIS AND HAL USER'S GUIDE

1. Introduction

CMSIS is the Cortex Microcontroller Software Interface Standard, and the Hardware Access Layer (HAL) is a defined part of this standard.

The HAL provides a layer of abstraction from the SiM3xxxx device registers. The functions and macros are non-blocking and simple; they cannot return error codes, so they are designed to never fail. The HAL is designed to abstract the individual bit fields of the module to a function name that describes the action the bit is controlling.

Note: HAL functions and macros are not designed to be thread-safe. These routines do not disable interrupts during non-monotonic register modifications.

The HAL sits one layer above the hardware and is the only code that accesses the registers directly. More complex firmware systems like a real time operating system (RTOS) or code example sit on top of the HAL and CMSIS.

Figure 1 shows the Precision32™ firmware layer block diagram.

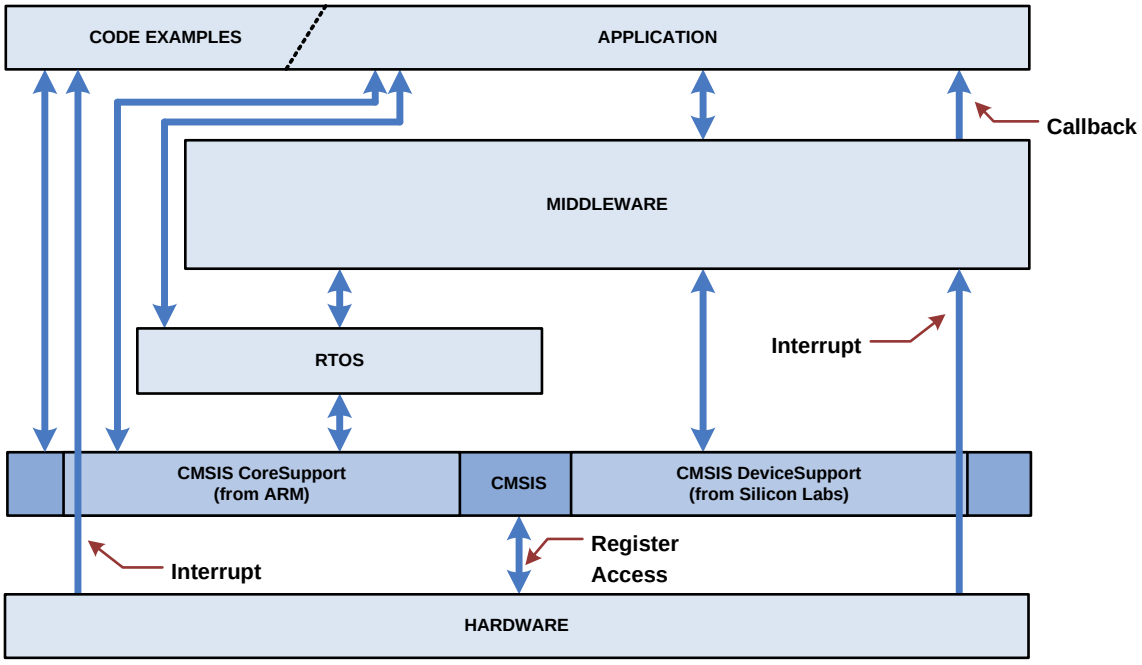

Figure 1. Firmware Layer Block Diagram

2. Peripheral Memory Organization

Each peripheral exists as a set of registers in memory. Most peripherals start at 0x1000 address blocks in the peripheral memory area starting at address 0x40000000. The base pointer of a peripheral points to the starting address of the peripheral, and each register is an offset from the base address. In the case of the USART0 module, the base pointer is SI32_USART_0, and it is assigned an address of 0x40000000, since it's the first peripheral in the peripheral memory area.

The registers each take 16 bytes (0x10) of memory: a word each for the register and the SET, CLR, and MSK addresses. These addresses are reserved for registers that do not implement them.

Figure 2 shows the USART0 registers as they appear in memory.

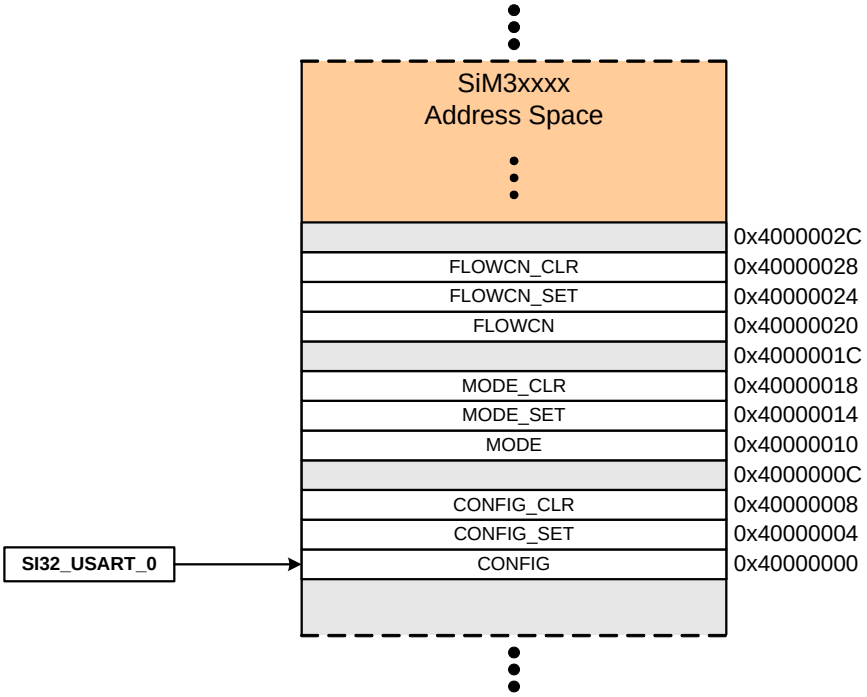

Figure 2. USART0 Registers in Memory

3. HAL Organization

The HAL is organized based on the SiM3xxxx peripheral modules. Modules of the same type and revision are exactly the same, so these modules share the same generic description. The individual instances of the modules then instantiate their own copies of the generic description. For example, the HAL implements a USART (module) A (revision) type. This type is then instantiated multiple times for USART0 and USART1, which have their own base pointers.

Each generic module has a *_Registers.h file that contains a module structure comprised of register structures and bit fields properly aligned in memory. The *_Type.h file contains the HAL interface for the module, and the *_Type.c contains the HAL implementation. Finally, there's a device header file named for the device (e.g., sim3u1xx.h) that contains base pointer and interrupt vector instantiations for each module on a device.

Figure 3 displays a block diagram showing the relationship of the HAL files.

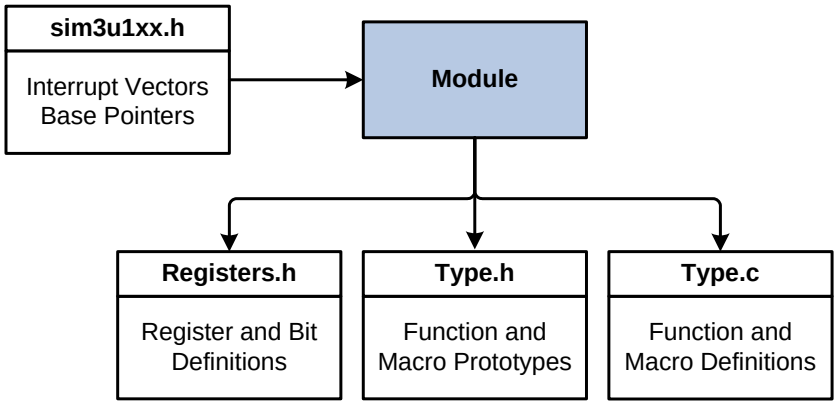

Figure 3. HAL Block Diagram

3.1. USART0 HAL Example

The sim3u1xx.h device file contains the USART0 interrupt vector (USART0_IRQn) and base pointer information (SI32_USART_0). The SI32_USART_A_Registers.h file contains the module structure, which includes structures of bit fields for each register. The SI32_USART_A_Type.h and Type.c files contain routines that access each of these bits.

Figure 4 displays a block diagram showing the relationship of the HAL files for the USART0 module.

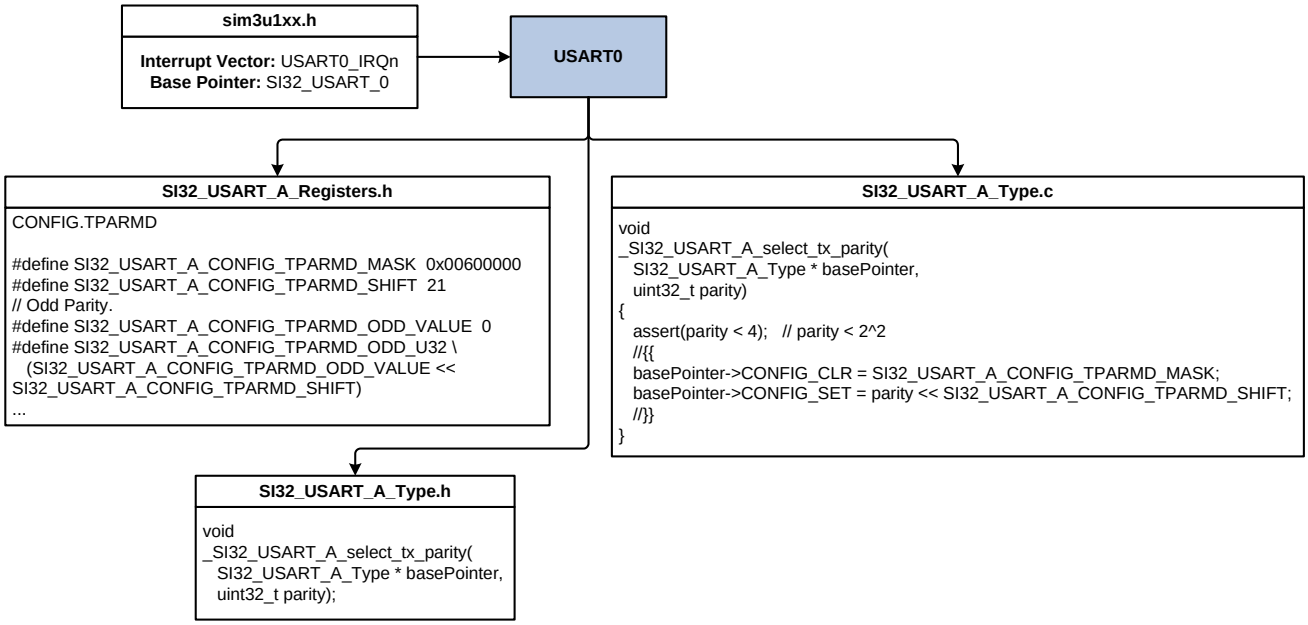

Figure 4. Example USART HAL Block Diagram

Firmware can call the USART0 select tx parity routine using the base pointer defined in si3mu1xx.h and the function implemented in SI32_USART_A_Type.c:

```
SI32_USART_A_select_tx_parity(SI32_USART_0, parity);
```

Figure 5 shows an example set of registers for the USART0 module.

Register Name	Title	Address (ALL Access)	SET (+0x4)	CLR(+0x8)	MSK (+0xC)
USART0 Registers—Bit Details in SiM3xxxx Reference Manual					
USART0_CONFIG	Module Configuration	0x4000_0000	Y	Y	
USART0_MODE	Module Mode Select	0x4000_0010	Y	Y	
USART0_FLOWCN	Flow Control	0x4000_0020	Y	Y	
USART0_CONTROL	Module Control	0x4000_0030	Y	Y	
USART0_IPDELAY	Inter-Packet Delay	0x4000_0040			
USART0_BAUDRATE	Transmit and Receive Baud Rate	0x4000_0050			
USART0_FIFOCN	FIFO Control	0x4000_0060	Y	Y	
USART0_FIFO	FIFO Input/Output Data	0x4000_0070			

Figure 5. Example USART Registers

Figure 6 illustrates the resulting USART module structure in Registers.h.

```
typedef struct SI32_USART_A_Struct
{
   struct SI32_USART_A_CONFIG_Struct      CONFIG      ; // Base Address + 0x0
   volatile uint32_t                      CONFIG_SET;
   volatile uint32_t                      CONFIG_CLR;
   uint32_t                               reserved0;
   struct SI32_USART_A_MODE_Struct        MODE        ; // Base Address + 0x10
   volatile uint32_t                      MODE_SET;
   volatile uint32_t                      MODE_CLR;
   uint32_t                               reserved1;
   struct SI32_USART_A_FLOWCN_Struct      FLOWCN      ; // Base Address + 0x20
   volatile uint32_t                      FLOWCN_SET;
   volatile uint32_t                      FLOWCN_CLR;
   uint32_t                               reserved2;
   struct SI32_USART_A_CONTROL_Struct     CONTROL     ; // Base Address + 0x30
   volatile uint32_t                      CONTROL_SET;
   volatile uint32_t                      CONTROL_CLR;
   uint32_t                               reserved3;
   struct SI32_USART_A_IPDELAY_Struct     IPDELAY     ; // Base Address + 0x40
   uint32_t                               reserved4;
   uint32_t                               reserved5;
   uint32_t                               reserved6;
   struct SI32_USART_A_BAUDRATE_Struct    BAUDRATE    ; // Base Address + 0x50
   uint32_t                               reserved7;
   uint32_t                               reserved8;
   uint32_t                               reserved9;
   struct SI32_USART_A_FIFOCN_Struct      FIFOCN      ; // Base Address + 0x60
   volatile uint32_t                      FIFOCN_SET;
   volatile uint32_t                      FIFOCN_CLR;
   uint32_t                               reserved10;
   struct SI32_USART_A_DATA_Struct        DATA        ; // Base Address + 0x70
   uint32_t                               reserved11;
   uint32_t                               reserved12;
   uint32_t                               reserved13;
} SI32_USART_A_Type;
```

Figure 6. Example USART Module Structure

Each of the registers has a corresponding structure that defines the bit fields within that register. In addition, registers that implement the clear and set addresses have 32-bit variables defined at the appropriate addresses.

Figure 7 shows the CONFIG register structure, which is declared as a part of the USART module structure.

```
struct SI32_USART_A_CONFIG_Struct
{
   union
   {
      struct
      {
         // Receiver Start Enable
         volatile uint32_t RSTRTEN: 1;
         // Receiver Parity Enable
         volatile uint32 t RPAREN: 1:
                        ●
                        ●
                        ●
         // Transmitter Stop Mode
         volatile uint32_t TSTPMD: 2;
         // Transmitter Parity Mode
         volatile uint32_t TPARMD: 2;
                  uint32_t reserved2: 1;
         // Transmitter Data Length
         volatile uint32_t TDATLN: 3;
                  uint32_t reserved3: 1;
                        ●
                        ●
                        ●
         // Transmitter Synchronous Mode Enable
         volatile uint32_t TSYNCEN: 1;
      };
      volatile uint32_t U32;
   };
};
```

Figure 7. Example USART CONFIG Register Structure

This CONFIG register structure has the TPARMD 2-bit field that controls the transmit parity. The U32 value declared at the bottom of the structure is a union with the bit fields and is an entity that firmware can use to access the entire register at one time.

The HAL Type.c functions and macros can access the entire USART0 CONFIG register:

```
SI32_USART_0->CONFIG.U32 = config;
```

The HAL can also read or write the TPARMD field in the USART0 CONFIG register:

```
parity = SI32_USART_0->CONFIG.TPARMD;

SI32_USART_0->CONFIG_SET = parity << SI32_USART_A_CONFIG_TPARMD_SHIFT;
```

Finally, the HAL can clear the TPARMD field:

```
SI32_USART_0->CONFIG_CLR = SI32_USART_A_CONFIG_TPARMD_MASK;
```

4. Performance

The HAL implements both macros and functions for most routines. This allows firmware layers that call the HAL to choose between the faster performance of macros or the smaller footprint of functions. For the routines that do not have an implemented macro, a macro still exists, but it just calls the function.

The functions and macros have the same parameters and names, but functions have an underscore prefix. For example, the function for setting the USART transmit parity is:

```
_SI32_USART_A_select_tx_parity(SI32_USART_0, parity);
```

The macro for the same routine is:

```
SI32_USART_A_select_tx_parity(SI32_USART_0, parity);
```

5. Revisions

Each version of the HAL sits in a separate folder. The path of these is **C:\SiLabs\32bit\si32\si32-x.y**, where **x** is the primary HAL version and **y** is the secondary HAL version. Each time a new version of the HAL is installed, it will leave all previous versions to eliminate the chance of a new install breaking a working firmware project.

In addition, any deprecated functions will remain a part of the HAL. These functions will either remain unchanged or call the new version to prevent the need to modify firmware when migrating to a newer version of the HAL.

6. Code Examples

Each version of the HAL includes stand-alone code examples for the device modules that use the macro routines by default. These examples can be found in **C:\SiLabs\32bit\si32-x.y\Examples** after installing the Precision32 IDE.

7. The HAL and AppBuilder

The Silicon Labs AppBuilder program uses HAL macros when configuring peripherals. The AppBuilder project options can select between different versions of the HAL.

8. Detailed Documentation

The detailed Silicon Labs CMSIS documentation can be found in the si32Cmsis Windows help file (si32Cmsis.chm). The documentation includes the Cortex-M3 Core Register Definitions, Core Function Interface, and Core Instruction Interface, as well as the SiM3xxxx HAL.

The si32Cmsis file shown in Figure 8 is installed in **C:\SiLabs\32bit\si32-x.y\Documentation** after installing the Precision32 IDE.

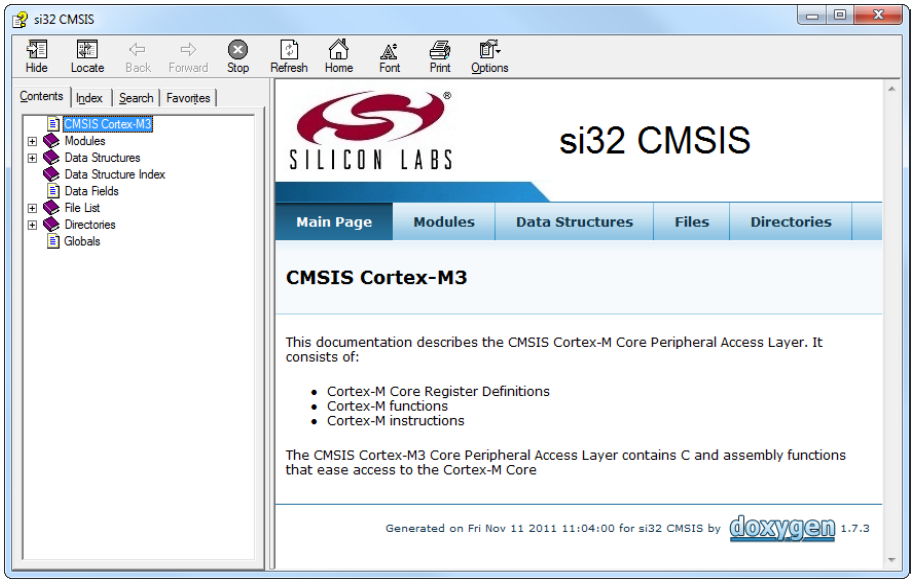

Figure 8. Silicon Labs HAL and CMSIS Documentation

NOTES:

CONTACT INFORMATION

Silicon Laboratories Inc.

400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page:
https://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.