VORAGO VA108x0 eFuse programming application note

November, 2016 Version 1.0

VA10800/VA10820

Abstract

The VA108x0 family of MCUs contains programmable eFuse memory that can be useful for board identification, serialization, sensor calibration information, product life tracking or other use cases where a relatively small amount of information is required to be permanently stored. Boot sequence variables can also be set by the eFuse memory.

The eFuse array can only be programmed and read via the JTAG port. During the boot sequence two 32-bit words are read from the eFuse and stored in a CPU register: EF_ID and EF_CONFIG.

This application note provides instructions and code to use one REB1 development board to program and read the eFuse array in a second VA108x0 MCU via the standard JTAG connection.

Table of contents

1 eFuse functionality 2
2 Hardware connection 3
3 Procedure to read and program eFuse memory 6
4 Running the example project 12
5 Conclusion 14
6 Other Resources 14

1 eFuse functionality

The eFuse memory is programmed by means of electromigration of the cobalt silicide on the fuse as opposed to an actual rupture of the fuse. The use of the electromigration process results in a more reliable fuse that does not rupture the fuse or the passivation and does not potentially contaminate the chip with metal debris that is ejected from the rupture site. The typical fusing current of a single bit and duration is 13mA for 200μs. A state machine inside the MCU programs a single bit at a time to limit the current draw.

Past space flights with different fuse type memory have shown the propensity for regrowth of the altered polysilicon resulting in a bit reverting from a programmed "1" to a "0". This regrowth is attributed to an extended time with a voltage potential existing between two closely spaced nodes in the memory cell. The VA108x0 only reads the eFuse memory after a RESET. Information from two locations are stored in registers, then the eFuse is powered down. By limiting the time that the voltage is applied to the memory array, the chance of any regrowth is virtually eliminated.

The VA108x0 contains a 32-bit by 32-entry eFuse array. The eFuse array is read during the RESET sequence and then turned off. Two of the 32-bit entries are stored in registers: EF_CONFIG and EF_ID. The other locations can only be read via the JTAG interface.

The EF_CONFIG register determines the following:

- Boot SPI clock rate (bus clock divided by 2, 6, 12 or 52)
- Boot SPI memory size (4k to 128 kbytes)
- Redundant boot (reads memory twice)
- Boot delay (0-500 mSec)
- ROM read instruction code (default is 0x3 which matches most SPI EEPROM, MRAM and FRAM memories)
- ROM address mode (16 or 24 bits, 128k memories require 24-bit addresses)

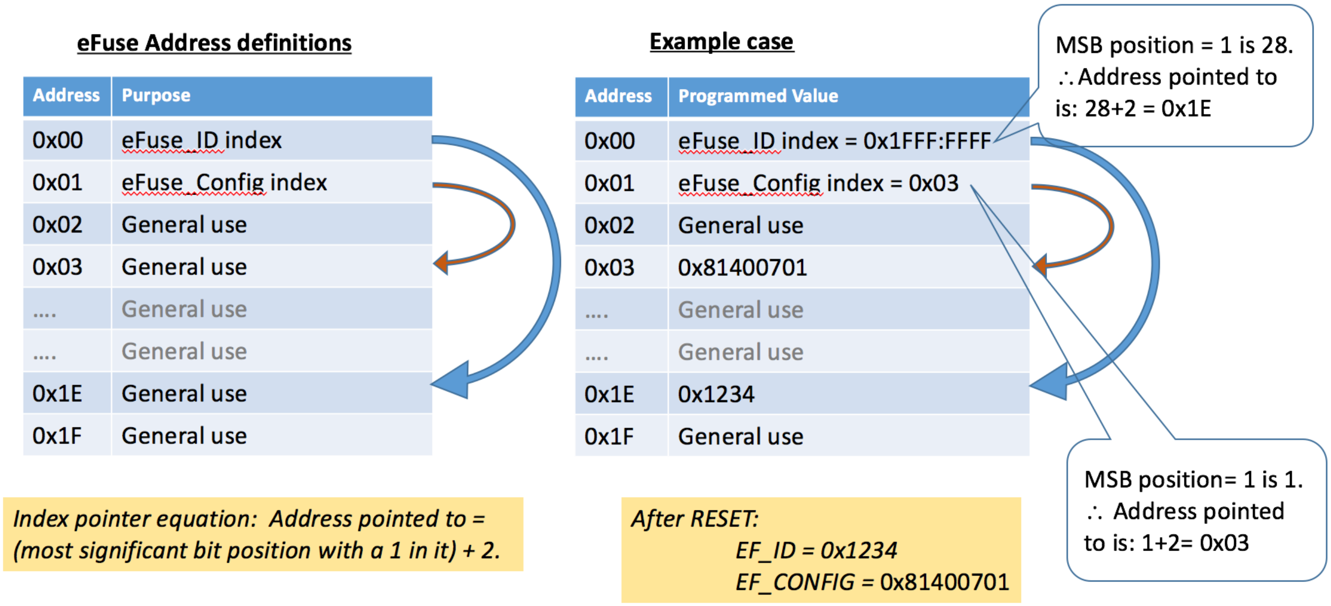
The eFuse array can be read and programmed via the JTAG interface. User code is not directly able to either read or program the eFuse memory. This restriction was implemented to ensure that the eFuse gate did not have a voltage potential across it for an extended time.

1.1 eFuse organization

There are 32 addresses each with 32 bits of information in the eFuse array. Addresses 0 & 1 have unique functionality and are used as index pointers to the EF_ID and EF_CONFIG fields. Since it is impossible to change a bit from a "1" to a "0", a unique pointer scheme is used whereby the most significant bit that is set to a "1" determines the index value per the following equation.

Index pointer location = (most significant bit position with a "1") + 2

See Figure 1 - eFuse Index pointer explanation for an illustration of this equation and an example case.



eFuse Address definitions

Address	Purpose
0x00	eFuse_ID index
0x01	eFuse_Config index
0x02	General use
0x03	General use
….	General use
….	General use
0x1E	General use
0x1F	General use

Index pointer equation: Address pointed to = (most significant bit position with a 1 in it) + 2.

Example case

Address	Programmed Value
0x00	eFuse_ID index = 0x1FFF:FFFF
0x01	eFuse_Config index = 0x03
0x02	General use
0x03	0x81400701
….	General use
….	General use
0x1E	0x1234
0x1F	General use

MSB position = 1 is 28.
∴Address pointed to is: 28+2 = 0x1E

MSB position= 1 is 1.
∴ Address pointed to is: 1+2= 0x03

After RESET:
EF_ID = 0x1234
EF_CONFIG = 0x81400701

Figure 1 - eFuse Index pointer explanation

Caution: Programming the wrong information into the eFuse array can leave the entire device inoperative. Care should be taken to double-check the planned programmed data prior to programming the device.

2 Hardware connection

The hardware connection from a REB1 board to another VA108x0 MCU is described in this section.

A convenient way to connect a REB1 development board to the standard 10-pin JTAG connector is to use a commercially available assembly from Olimex (ARM-JTAG-20-10) which retails for less than $10 USD. The standard JTAG connector is 0.05" center x 2 row. A commonly used part is the Samtec FTSH male vertical box header- part # SHF-105-01-L-D-TH. It is also possible to fly wire the connections or to make a custom cable for this purpose. Keeping the cable under 6 inches in length is important for signal integrity.

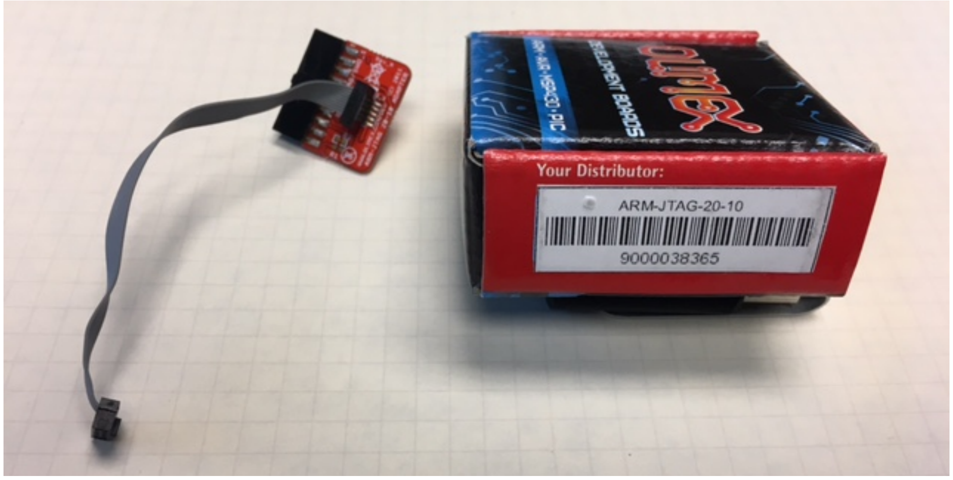


Figure 2 - Olimex harness for connecting to standard 0.05" x 2 row JTAG interface

The harness plugs directly into J14 of the REB1 board for the digital signals and a separate ground connection must be made as shown in Figure 3. Table 1 shows the connections from the REB1 board to the Olimex cable. Note that pin 1 of J14 aligns with pin 19 of the Olimex cable's 20-pin header. An additional ground connection is required to have both boards at the same potential. Figure 4 shows the Olimex cable assembly schematic.

Once the connector is in place you are ready to run software on the REB1 board to program and read the eFuse memory. Note that both boards will require their own power source since the JTAG interface does not have a power line.

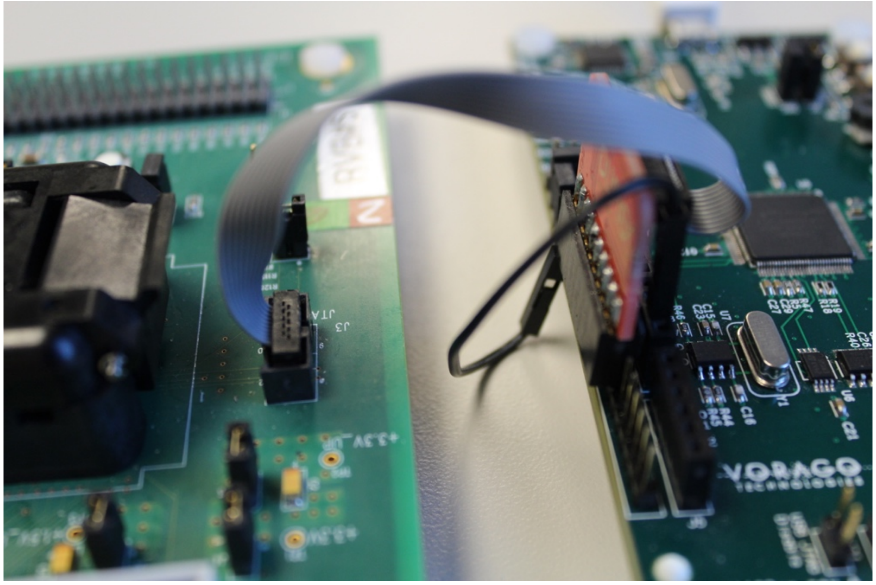


Figure 3 – Picture of REB1 board to JTAG connector on second board

Olimex 20 pin connector		REB1- J14 connector		Notes
Pin #	Signal	Pin #	Signal	
19	+5V	1	PORTB[10]	Not used – left as input
17	NC	2	PORTB[11]	Not used – left as input
15	RST	3	PORTB[12]	Board with MCU being programmed must have this signal tied to TRSTn
13	TDO	4	PORTB[13]	MCU input
11	NC	5	PORTB[14]	
9	TCK	6	PORTB[15]	MCU output
7	TMS	7	PORTB[16]	MCU output
5	TDI	8	PORTB[17]	MCU output
3	TRST	9	PORTB[18]	No connect on cable; Not used – left as input
1	VTRef	10	PORTB[19]	Not used – left as input
Note: A ground connection is required between boards. We recommend using a jumper wire from a REB1 ground pin to pin 20 of the Olimex 20-pin connector. This will minimize the ground loop area between boards.				

Table 1 - Connections between REB1 and JTAG connector

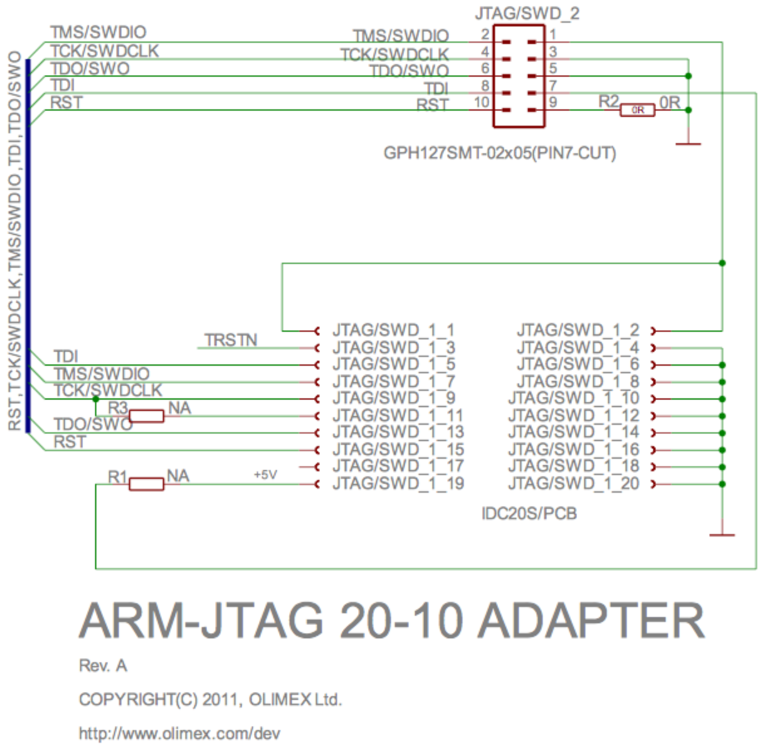


Figure 4 - Olimex JTAG harness schematic

3 Procedure to read and program eFuse memory

The VA108x0 MCU has two JTAG Test Access Ports (TAPS). They are in series as shown in Figure 5. The first is for the standard ARM® debug TAP (DBGTAP) which has a four-bit instruction register. The second is the VA108x0 test port which has a five-bit instruction register and is unique to the device. The second TAP contains the interface to the eFuse block.

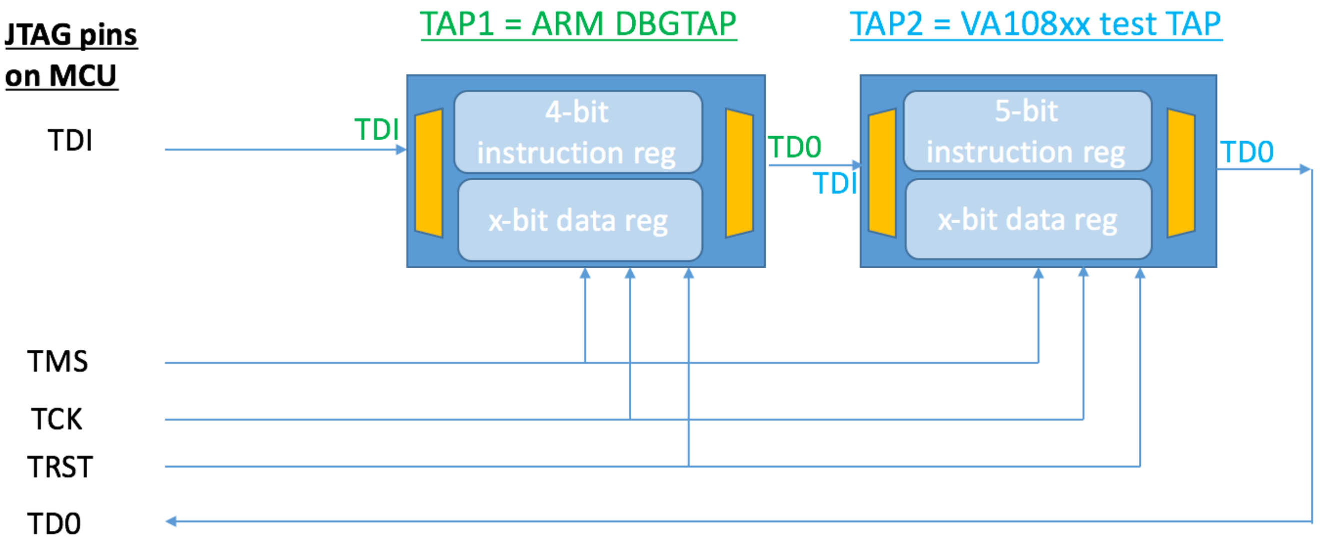


Figure 5 - Serial Test Access Port connections

To not have TAP1 interfere with communications to TAP2, we will always send four zeros to the TAP1 instruction register. This keeps TAP1 in bypass mode. For example, to select the instruction register 0x17 of the VA108x0 TAP, we would send nine bits on TDI to the JTAG instruction register. The first four bits would be zeros and the last five would be 0x17 or %10111. Note that the JTAG convention is to shift the least significant bit out first so a logic analyzer would show %111010000 for this transaction.

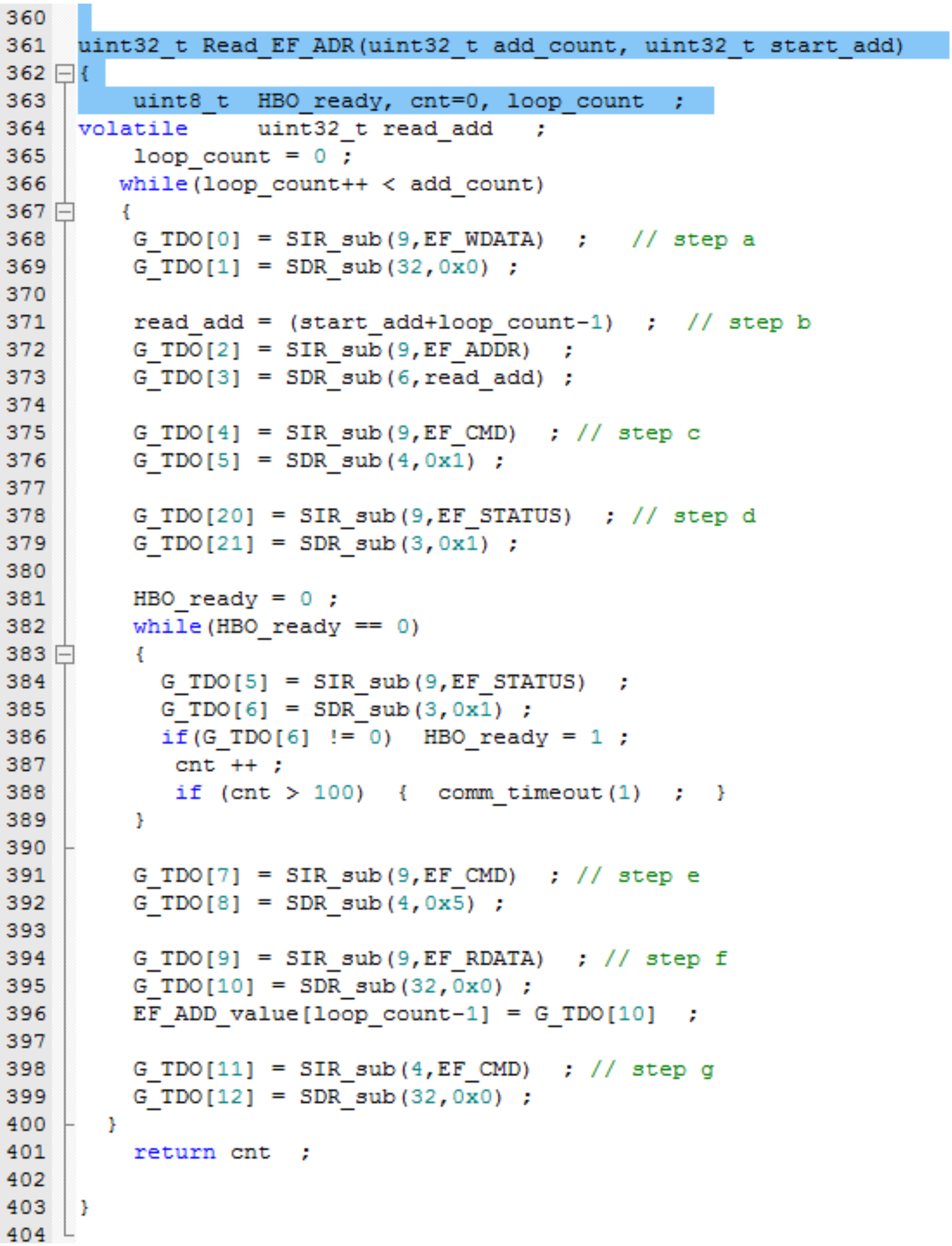
Section 5 of the VA108x0 programmers guide has details on the eFuse read and programming steps as outlined in the following two sections.

3.1 Reading an eFuse location

The following steps must be taken to properly read eFuse memory. Each of these steps is conducted by first shifting data to the instruction register (SIR) followed by either a read or write of the associated data register via the shift data register (SDR) command. Data registers have variable length and it is only required to shift the number of bits implemented. As shown in Figure 6, the software accompanying the application note has comments to make it easy to follow the steps being taken.

a) Load EF_WDATA register with 0 (Clear Test Read Data)
b) Load EF_ADDR register with the desired read address (Load Address)
c) Load EF_CMD registers with 0x1 (Interface Enable)

d) Poll/Read EF_STATUS for Bit 0 being 1 (Oscillator is running)
e) Load EF_CMD registers with 0x5 (Issue Read Command)
f) Read EF_RDATA register (Result of EFuse read)
g) Load EF_CMD registers with 0x0 (Interface Disable)



```

uint32_t Read_EF_ADR(uint32_t add_count, uint32_t start_add)
{
    uint8_t   HBO_ready, cnt=0, loop_count  ;
volatile     uint32_t read_add   ;
    loop_count = 0 ;
   while(loop_count++ < add_count)
   {
    G_TDO[0] = SIR_sub(9,EF_WDATA)  ;   // step a
    G_TDO[1] = SDR_sub(32,0x0) ;

    read_add = (start_add+loop_count-1)  ;  // step b
    G_TDO[2] = SIR_sub(9,EF_ADDR)  ;
    G_TDO[3] = SDR_sub(6,read_add) ;

    G_TDO[4] = SIR_sub(9,EF_CMD)  ; // step c
    G_TDO[5] = SDR_sub(4,0x1) ;

    G_TDO[20] = SIR_sub(9,EF_STATUS)  ; // step d
    G_TDO[21] = SDR_sub(3,0x1) ;

    HBO_ready = 0 ;
    while(HBO_ready == 0)
    {
      G_TDO[5] = SIR_sub(9,EF_STATUS)  ;
      G_TDO[6] = SDR_sub(3,0x1) ;
      if(G_TDO[6] != 0)  HBO_ready = 1 ;
       cnt ++ ;
       if (cnt > 100)  {  comm_timeout(1)  ;  }
    }

    G_TDO[7] = SIR_sub(9,EF_CMD)  ; // step e
    G_TDO[8] = SDR_sub(4,0x5) ;

    G_TDO[9] = SIR_sub(9,EF_RDATA)  ; // step f
    G_TDO[10] = SDR_sub(32,0x0) ;
    EF_ADD_value[loop_count-1] = G_TDO[10]  ;

    G_TDO[11] = SIR_sub(4,EF_CMD)  ; // step g
    G_TDO[12] = SDR_sub(32,0x0) ;
  }
    return cnt  ;

}

```

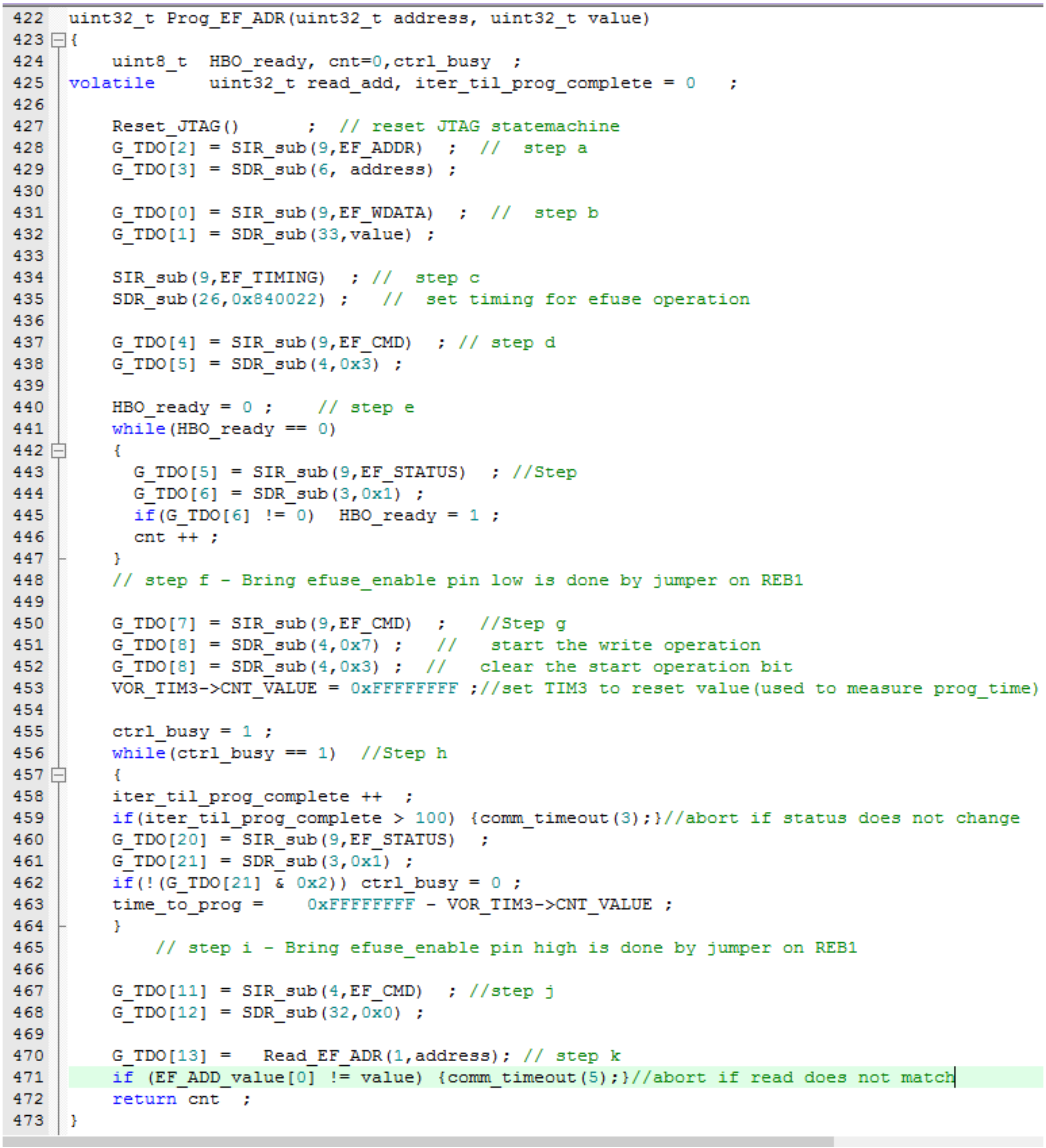
Figure 6 - Subroutine to read eFuse memory

3.2 Writing an eFuse location

Similar to reading an eFuse location, a set of steps must be taken to program eFuse memory. As shown in Figure 7, the software accompanying the application note has comments showing each step to make it easy to follow the process.

a) Load EF_ADDR register with the desired write address (Load Address)
b) Load EF_WDATA register with data (Load Data)
c) Load EF_TIMING register with proper timing data
d) Load EF_CMD registers with 0x3 (Interface Enable with Write Mode)
e) Poll/Read EF_STATUS for Bit 0 being 1 (Oscillator is running)
f) Bring EFUSE_WRITE_ENn pin low (Enable Write)
g) Load EF_CMD registers with 0x7 (Issue Write Command)
h) Poll/Read EF_STATUS register for Bit 1 being 0 (Check/Wait for write complete)
i) Bring EFUSE_WRITE_ENn pin high (Disable Write)
j) Load EF_CMD registers with 0x0 (Interface Disable)
k) Do eFuse Read procedure to verify result

Note: Steps f and i are not implemented in software since there is a jumper on the board that is being programmed. It is possible to read the eFuse memory with the EFUSE_WRITE_ENn pin in either state. There is no need to stop the program after step h to remove the jumper controlling the EFUSE_WRITE_ENn pin.



```
uint32_t Prog_EF_ADR(uint32_t address, uint32_t value)
{
    uint8_t  HBO_ready, cnt=0,ctrl_busy  ;
volatile     uint32_t read_add, iter_til_prog_complete = 0   ;

    Reset_JTAG()       ;  // reset JTAG statemachine
    G_TDO[2] = SIR_sub(9,EF_ADDR)  ;  //  step a
    G_TDO[3] = SDR_sub(6, address) ;

    G_TDO[0] = SIR_sub(9,EF_WDATA)  ;  //  step b
    G_TDO[1] = SDR_sub(33,value) ;

    SIR_sub(9,EF_TIMING)  ; //  step c
    SDR_sub(26,0x840022) ;   //  set timing for efuse operation

    G_TDO[4] = SIR_sub(9,EF_CMD)  ; // step d
    G_TDO[5] = SDR_sub(4,0x3) ;

    HBO_ready = 0 ;     // step e
    while(HBO_ready == 0)
    {
      G_TDO[5] = SIR_sub(9,EF_STATUS)  ; //Step
      G_TDO[6] = SDR_sub(3,0x1) ;
      if(G_TDO[6] != 0)  HBO_ready = 1 ;
      cnt ++ ;
    }
    // step f - Bring efuse_enable pin low is done by jumper on REB1

    G_TDO[7] = SIR_sub(9,EF_CMD)  ;   //Step g
    G_TDO[8] = SDR_sub(4,0x7) ;   //   start the write operation
    G_TDO[8] = SDR_sub(4,0x3) ;  //   clear the start operation bit
    VOR_TIM3->CNT_VALUE = 0xFFFFFFFF ;//set TIM3 to reset value(used to measure prog_time)

    ctrl_busy = 1 ;
    while(ctrl_busy == 1)  //Step h
    {
    iter_til_prog_complete ++  ;
    if(iter_til_prog_complete > 100) {comm_timeout(3);}//abort if status does not change
    G_TDO[20] = SIR_sub(9,EF_STATUS)  ;
    G_TDO[21] = SDR_sub(3,0x1) ;
    if(!(G_TDO[21] & 0x2)) ctrl_busy = 0 ;
    time_to_prog =    0xFFFFFFFF - VOR_TIM3->CNT_VALUE ;
    }
        // step i - Bring efuse_enable pin high is done by jumper on REB1

    G_TDO[11] = SIR_sub(4,EF_CMD)  ; //step j
    G_TDO[12] = SDR_sub(32,0x0) ;

    G_TDO[13] =   Read_EF_ADR(1,address); // step k
    if (EF_ADD_value[0] != value) {comm_timeout(5);}//abort if read does not match
    return cnt  ;
}
```

Figure 7 - Subroutine to program eFuse memory

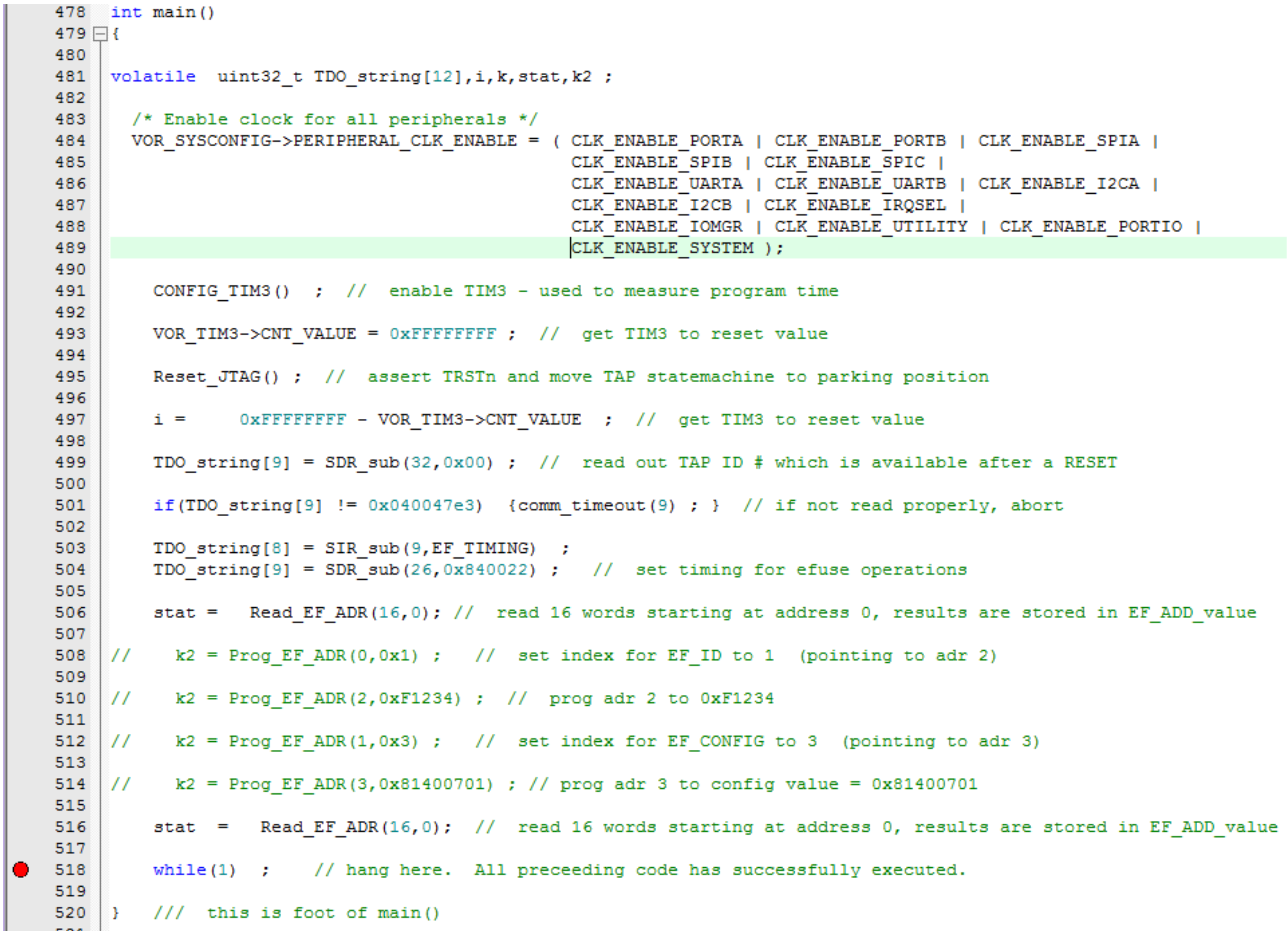
4 Running the example project

The example code was developed in the Keil MDK environment. Please follow the instructions in the REB1 User's manual to install the IDE and the Vorago Pack file. Once that is done, simply click on the project file called "JTAG_va108xx.uvprojx".

The main routine as shown in Figure 8, has a call to read the first 16 words of eFuse array and store the data in an array "EF_ADD_value[]". The sample project will already have a memory window active showing this array. The calls to program the eFuse on lines 508 - 514 have been commented out to prevent inadvertent writes to memory. When the locations and contents have been determined, remove the "//" at the beginning of these lines and modify the address and data values. Then run the program.

If the program and / or read operation is successful, the code will hang at a while(1) statement on line 518 as shown in Figure 8. This statement should already have a breakpoint set in the provided project as denoted by the red circle in the left margin.

If another MCU is not connected or the programming operation fails, the code will hang in the comm_timeout() subroutine. The provided project will have a breakpoint set in this routine.



```
int main()
{

volatile  uint32_t TDO_string[12],i,k,stat,k2 ;

  /* Enable clock for all peripherals */
  VOR_SYSCONFIG->PERIPHERAL_CLK_ENABLE = ( CLK_ENABLE_PORTA | CLK_ENABLE_PORTB | CLK_ENABLE_SPIA |
                                           CLK_ENABLE_SPIB | CLK_ENABLE_SPIC |
                                           CLK_ENABLE_UARTA | CLK_ENABLE_UARTB | CLK_ENABLE_I2CA |
                                           CLK_ENABLE_I2CB | CLK_ENABLE_IRQSEL |
                                           CLK_ENABLE_IOMGR | CLK_ENABLE_UTILITY | CLK_ENABLE_PORTIO |
                                           CLK_ENABLE_SYSTEM );

    CONFIG_TIM3()  ;  //  enable TIM3 - used to measure program time

    VOR_TIM3->CNT_VALUE = 0xFFFFFFFF ;  //  get TIM3 to reset value

    Reset_JTAG() ;  //  assert TRSTn and move TAP statemachine to parking position

    i =     0xFFFFFFFF - VOR_TIM3->CNT_VALUE  ;  //  get TIM3 to reset value

    TDO_string[9] = SDR_sub(32,0x00) ;  //  read out TAP ID # which is available after a RESET

    if(TDO_string[9] != 0x040047e3)  {comm_timeout(9) ; }  // if not read properly, abort

    TDO_string[8] = SIR_sub(9,EF_TIMING)  ;
    TDO_string[9] = SDR_sub(26,0x840022) ;   //  set timing for efuse operations

    stat =   Read_EF_ADR(16,0); //  read 16 words starting at address 0, results are stored in EF_ADD_value

//    k2 = Prog_EF_ADR(0,0x1) ;   //  set index for EF_ID to 1  (pointing to adr 2)

//    k2 = Prog_EF_ADR(2,0xF1234) ;  //  prog adr 2 to 0xF1234

//    k2 = Prog_EF_ADR(1,0x3) ;   //  set index for EF_CONFIG to 3  (pointing to adr 3)

//    k2 = Prog_EF_ADR(3,0x81400701) ; // prog adr 3 to config value = 0x81400701

    stat  =   Read_EF_ADR(16,0);  //  read 16 words starting at address 0, results are stored in EF_ADD_value

    while(1)  ;    // hang here.  All preceeding code has successfully executed.

}   ///  this is foot of main()
```

Figure 8 - Main routine from sample project

5 Conclusion

This application note has explained some of the potential use cases for the eFuse array on the VA108x0 MCU. Example hardware and software was explained that can be used to program one VA108x0 from a separate REB1 development board. A user should now be able to fully utilize the eFuse memory array on the VA108x0 MCU.

6 Other Resources

Vorago VA108x0 programmers guide:
http://www.voragotech.com/sites/default/files/VA10800_VA10820_PG_July2016revision1.16%5B4%5D.pdf

Vorago MCU products: http://www.voragotech.com/vorago-products

Vorago Application notes: http://www.voragotech.com/resources

Olmex connector schematic: https://www.olimex.com/Products/ARM/JTAG/ARM-JTAG-20-10/resources/ARM-JTAG-20-10-schematic.pdf.

1149.1-2013 - IEEE Standard for Test Access Port and Boundary-Scan Architecture:
http://standards.ieee.org/findstds/standard/1149.1-2013.html