Using hardware and software to make new stuff

Search

type, hit enter

Categories

- 3D Printing (1)
- Affinity Photo (1)
- Aide-memoir (1)
- Analog (1)
- Ansible (3)
- Dates to Remember (3)
- Docker (1)
- Electronics (144)
- ESP8266 (12)
- FPGA (4)
- Garden (1)
- General (8)
- Home Built CPU (6)
- Informatica (1)
- Internet of Things (8)
- iOS (2)
- KiCad (4)
- MSP430 (2)
- Netduino (49)
- NuttX (9)
- Photography (3)
- Pico (10)
- Raspberry Pi (17)
- Silverlight (6)
- Software Development (88)
- STM32 (5)
- STM8 (54)
- Tips (5)

Archives

- July 2024
- June 2024
- May 2024
- April 2024
- March 2024
- February 2024
- January 2024
- December 2023
- November 2023
- October 2023

- February 2020
- January 2020
- July 2019
- February 2018
- July 2017
- June 2017
- April 2017
- March 2017
- February 2017
- October 2016
- September 2016
- July 2016
- June 2016
- May 2016
- April 2016
- March 2016
- January 2016
- December 2015
- November 2015
- October 2015
- August 2015
- June 2015
- May 2015
- April 2015
- March 2015
- February 2015
- January 2015
- December 2014
- October 2014
- September 2014
- August 2014
- July 2014
- June 2014
- May 2014
- March 2014
- February 2014
- January 2014
- December 2013
- November 2013
- October 2013
- September 2013
- August 2013
- July 2013
- June 2013
- May 2013
- April 2013
- March 2013
- January 2013
- November 2012
- October 2012
- September 2012
- August 2012
- July 2012
- June 2012
- May 2012

- October 2011
- September 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- April 2010

Storing Data in EEPROM on the STM8S

During a recent project it became desirable to store a small amount of data in some non-volatile memory in order that the system state could be restored following loss of power. This article demonstrates how to achieve this by writing a small amount of data to the data EEPROM of the STM8S105Cr micro-controller on the STM8S Discovery board.

Memory Layout, Access and Protection

The Data EEPROM area of the STM8S series of micro-controllers varies depending upon the specific unit being used. For this article the specific micro-controller being used is the STM8S105C6 on the STM8S Discovery board. This micro-controller has 1KByte of EEPROM in the address range 0x4000 – 0x43ff.

By default the data area is write protected and cannot be modified by the main program. The write protection is removed by using a key to unlock the EEPROM data area. The flash program area is similarly protected but we will only consider the data EEPROM area. In order to write to the EEPROM area the application will need to write two security keys called Memory Access Security System (MASS) keys to the FLASH_DUKR register. These keys will unlock the EEPROM area and allow the application to write data to the EEPROM until the application turns write protection back on.

The MASS keys for the data EEPROM area are:

- 0xae
- 0x56

  The algorithm for enabling write access to the EEPROM data area is as follows:

  - Check the DUL bit of FLASH_IASPR. If this bit is set then the data area is writeable and no further action is required.
  - Write the first MASS key (0xae) to the FLASH_DUKR register.
  - Write the second MASS key (0x56) to the FLASH_DUKR register.

  At the end of the process of successfully writing the MASS keys the DUL bit of the FLASH_IASPR register will be set. This bit will remain set until either the application changes the bit or the micro-controller is reset. Resetting the DUL bit programatically reinstates the write protection for the EEPROM memory.

Read-while-write (RWW)

This feature is not available on all of the STM8S family of processors and you should consult the data sheet for the unit being used if you are interested in using this feature. The RWW feature allows the program memory to be read whilst the EEPROM memory is being written to.

individual bytes into the EEPROM memory. To erase a byte simply write 0x00 into the memory location.

Word and Page Programming

The STM8S also allows word (4 bytes) and block programming. Both of these are faster than byte programming and block programming is faster than word programming. These features will not be discussed further here and are mentioned simply for awareness.

Interrupts

The system can be configured to generate an interrupt for the following event:

- Successful write operation.
- Successful erase operation.
- Illegal operation (writing to protected pages).

Interrupts are enabled by setting FLASH_CR1_IE to 1. When this bit is set, an interrupt will be generated when the FLASH_IASPR_EOP or FLASH_IASPR_WR_PG_DIS bits are set.

Software

The original aim of the software was to write a small amount of data to the data EEPROM of the STM8S and use the ST Visual Programmer to verify that the memory contents had changed. As the project progressed it became apparent that we would also need some code to verify the data.

Writing the Data

The application to write data to the EEPROM is relatively simple:

```
//
//  Write a series of bytes to the EEPROM of the STM8S105C6.
//
//  This software is provided under the CC BY-SA 3.0 licence.  A
//  copy of this licence can be found at:
//
//  http://creativecommons.org/licenses/by-sa/3.0/legalcode
//
#if defined DISCOVERY
    #include <iostm8S105c6.h>
#else
    #include <iostm8s103f3.h>
#endif

//
//  Data to write into the EEPROM.
//
unsigned int _pulseLength[] = { 2000U, 27830U, 400U, 1580U, 400U, 3580U,
unsigned char _onOrOff[] =    {   1,      0,     1,     0,    1,     0,
char numberOfValues = 7;

//-----------------------------------------------------------------------
//
//  Write the default values into EEPROM.
//
```

```c
    //
    if (FLASH_IAPSR_DUL == 0)
    {
        FLASH_DUKR = 0xae;
        FLASH_DUKR = 0x56;
    }
    //
    //  Write the data to the EEPROM.
    //
    char *address = (char *) 0x4000;        //  EEPROM base address.
    *address++ = (char) numberOfValues;
    for (int index = 0; index < numberOfValues; index++)
    {
        *address++ = (char) (_pulseLength[index] & 0xff);
        *address++ = (char) ((_pulseLength[index] >> 8) & 0xff);
        *address++ = _onOrOff[index];
    }
    //
    //  Now write protect the EEPROM.
    //
    FLASH_IAPSR_DUL = 0;
}

//----------------------------------------------------------------------
//
//  Main program loop.
//
void main()
{
    SetDefaultValues();
}
```

The application simply enables writing to the EEPROM and then writes data to the memory. It also re-enables the write protection at the end of the write operation.

Verifying the Data

Testing this application should simply be a case of creating a new project, putting the above in *main.c*, setting some options and then running the code. The EEPROM data can then be read by *ST Visual Develop* and verified by hand. After compiling and executing the above code, start *ST visual Programmer*, connect it to the STM8S Discovery board and download the contents of the EEPROM:

This does not look correct. Double checking the code against *RM0016 – Reference Manual* all looks good with the application. So try downloading the EEPROM data again:

This time the data looks good and the values appear to be correct.

Downloading the EEPROM data again gave the first set of results. Trying for a fourth thime gave the second set of results. It appears that the correct data is only retrieved every second attempt (for reference, I am using *ST Visual Develop* version 3.2.8 on Windows 8).

At this point I decided that the only way to ensure that the data is in fact correct is to write a verification method into the code. The new application becomes:

```c
//
```

```
//  copy of this licence can be found at:
//
//  http://creativecommons.org/licenses/by-sa/3.0/legalcode
//
#if defined DISCOVERY
    #include <iostm8S105c6.h>
#else
    #include <iostm8s103f3.h>
#endif

//
//  Data to write into the EEPROM.
//
unsigned int _pulseLength[] = { 2000U, 27830U, 400U, 1580U, 400U, 3580U
unsigned char _onOrOff[] =    {   1,      0,     1,     0,    1,     0,
char numberOfValues = 7;

//--------------------------------------------------------------------
//
//  Write the default values into EEPROM.
//
void SetDefaultValues()
{
    //
    //  Check if the EEPROM is write-protected.  If it is then unlock t
    //
    if (FLASH_IAPSR_DUL == 0)
    {
        FLASH_DUKR = 0xae;
        FLASH_DUKR = 0x56;
    }
    //
    //  Write the data to the EEPROM.
    //
    char *address = (char *) 0x4000;        //  EEPROM base address.
    *address++ = (char) numberOfValues;
    for (int index = 0; index < numberOfValues; index++)
    {
        *address++ = (char) (_pulseLength[index] & 0xff);
        *address++ = (char) ((_pulseLength[index] >> 8) & 0xff);
        *address++ = _onOrOff[index];
    }
    //
    //  Now write protect the EEPROM.
    //
    FLASH_IAPSR_DUL = 0;
}

//--------------------------------------------------------------------
//
//  Verify that the data in the EEPROM is the same as the data we
//  wrote originally.
//
void VerifyEEPROMData()
{
    PD_ODR_ODR2 = 1;            //  Checking the data
    PD_ODR_ODR3 = 0;            //  No errors.
```

```
        PD_ODR_ODR3 = 1;
    }
    else
    {
        for (int index = 0; index < numberOfValues; index++)
        {
            unsigned int value = *address++;
            value += (*address++ << 8);
            if (value != _pulseLength[index])
            {
                PD_ODR_ODR3 = 1;
            }
            if (*address++ != _onOrOff[index])
            {
                PD_ODR_ODR3 = 1;
            }
        }
    }
    PD_ODR_ODR2 = 0;        // Finished processing.
}

//---------------------------------------------------------------------
//
//  Setup port D for data output.
//
void SetupPorts()
{
    //
    //  Initialise Port D.
    //
    PD_ODR = 0;             //  All pins are turned off.
    PD_DDR = 0xff;          //  All bits are output.
    PD_CR1 = 0xff;          //  All pins are Push-Pull mode.
    PD_CR2 = 0xff;          //  Pins can run up to 10 MHz.
}

//---------------------------------------------------------------------
//
//  Main program loop.
//
void main()
{
    SetupPorts();
    SetDefaultValues();
    VerifyEEPROMData();
}
```

The application uses Port D, pins 2 and 3 to indicate how the verification is proceeding. Pin D2 goes high when the application is verifying the data. Pin D3 is used to indicate if an error is found. Compiling the above and connecting up a scope gives the following output:

Pin D2 is connected to the yellow channel and pin D3 is connected to the blue channel. The above shows that the verification process starts and no errors are generated.

Conclusion

In it's simplest form, the code required to store the data is trivial only requiring the developer to enable the write operations and then disable after the data has been written successfully.

In addition to the above I would recommend that some form of checksum value is written into the EEPROM as it is possible that the power is lost as the data is being written into the EEPROM. In this case there are two arrays being written and we may only have written half of the data when the power is lost. This is left as an exercise for the reader.

Tags: Electronics, STM8, The Way of the Register

Thursday, September 12th, 2013 at 8:35 am • Electronics, STM8 • RSS 2.0 feed Both comments and pings are currently closed.

Comments are closed.

Pages

- About
- Making a Netduino GO! Module
- The Way of the Register
- Making an IR Remote Control
- Weather Station Project
- NuttX and Raspberry Pi PicoW

Support This Site

Buy me a coffee

Currently Reading