Adding 32 KB of Serial SRAM to a Stellaris® Microcontroller

Application Note

AN01243-02

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Table of Contents

Introduction 4
Electrical Connections 4
Practical Issues.......... 4
Example Application 5
Example Source Code.......... 6
Conclusion 13
References 13

Introduction

This application note describes how to interface an AMI Semiconductor, Inc., 32 KB serial SRAM device to a Stellaris® microcontroller using a Synchronous Serial Interface (SSI). It contains example source code and an examination of practical implementation issues.

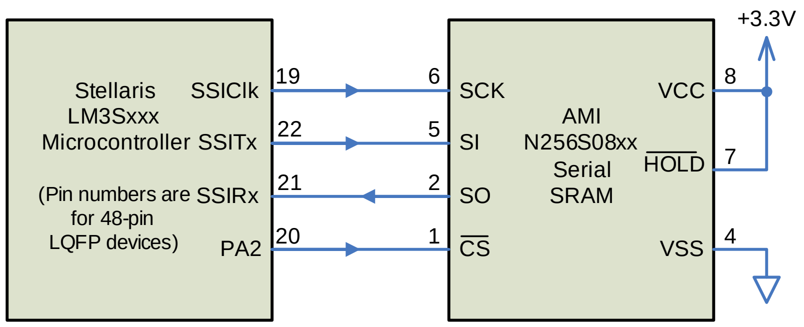
Although Stellaris microcontrollers have generous internal SRAM capabilities, certain applications may have data storage requirements that exceed the 8 KB limit of the Stellaris LM3S8xx series devices. Since microcontrollers do not have an external parallel data-bus, serial memory options must be considered. Until recently, the ubiquitous serial EEPROM/flash device was the only serial memory solution. The major limitations of EEPROM and flash technology are slow write speed, slow erase times, and limited write/erase endurance.

Recently, serial SRAM devices have become available as a solution for high-speed data applications. The N256S08xxHDA series of devices, from AMI Semiconductor, offer 32 K x 8 bits of low-power data storage, a fast Serial Peripheral Interface (SPI) serial bus, and unlimited write cycles. The parts are available in 8-pin SOIC and compact TSSOP packages.

Electrical Connections

The AMI serial SRAM connects directly to any LM3Sxxx device's SSI port. No discrete components are necessary to complete the interface.

Figure 1. Wiring a Stellaris MCU to a Serial SRAM



This implementation does not require the use of the serial SRAM's hold feature. The hold signal is typically used if there is more than one device on the SSI and the application needs to interrupt a serial SRAM access with another bus transaction. If needed, the nHOLD signal could be routed to a Stellaris GPIO pin.

The serial clock (SCK) signal is normally low with data (SI, SO) being sampled on the rising edge. This corresponds to the Motorola SPI Mode 0 on Stellaris Microcontrollers.

The AMI serial SRAM has low operating and standby power. Our tests confirmed the standby current (SPI inactive) at 1 uA and operating current (SPI active with 4 MHz clock) at just 825 uA.

Practical Issues

The first and most obvious issue with serial SRAM is that, because it is not memory mapped, all data operations require a certain amount of microprocessor SRAM. The second consideration is access speed which, not surprisingly, is up to 500 times slower than internal SRAM.

Neither issue presents a problem when migrating a design from serial EEPROM to serial SRAM. Read times are comparable to EEPROMs while write times are significantly faster. Both are entirely adequate for applications that need to store configuration and calibration data.

That said, serial SRAM is ideal for storing large data structures that are used infrequently, accessed sequentially, or require battery back-up. For sequential use, such as storing serial communication packets, serial SRAM can accommodate data rates in excess of 250 kB/second.

Table 1. Serial SRAM Access Times with 4 MHz Clock

Serial SRAM Access Type	Write Time (us)	Read Time (us)
Random Access of 1-byte (“char”)	9	9
Random Access of 2-byte (“short”)	11	11
Sequential Access of 32 bytes	78	78
Sequential Access of 1 KB	2310	2310
Sequential Access of 32 KB	73734	73734

The second application area is as a supplement to internal SRAM. Speed and memory map issues mean that serial SRAM may not always be a panacea for an application where internal SRAM is exhausted. However, the example source code and Table 1 show that serial SRAM access is fast enough to be used for routine variable storage.

Example Application

The “Example Source Code” section on page 6 contains example source code for a Stellaris EKK811 Evaluation Kit which has been wired to an AMI Semiconductor 32 KB serial SRAM. The software runs a continuous write/read/verify test which displays the test count and result on the OLED display.

The example code initializes the SSI module in master mode with a 4 MHz clock—the fastest allowed by Stellaris devices for bi-directional communication.

The serial SRAM device is then configured for 8-bit burst mode operation. In burst mode, any number of bytes can be read or written once the command and base address have been sent. This is the most efficient method for quickly accessing large data structures.

The example code also shows how to use serial SRAM to store static variables that would otherwise consume internal SRAM. The *UpdateTestCounter()* function tracks how many test cycles have been completed. Without a serial SRAM, the counter variable would be defined as ‘static unsigned char’ and would require a byte of internal SRAM. By defining the counter as ‘unsigned char’, the counter can be read into an ARM core register from serial SRAM.

Before the counter variable was moved to serial SRAM, the example required 384 bytes of internal SRAM (compiled with ARM RealView compiler). With the counter in serial SRAM, the example requires 376 bytes of internal SRAM. In fact, the compiler was able to save 8 bytes because of padding between data sections and the stack. This simple example demonstrates that even small variables can be moved from internal SRAM to serial SRAM. It can be easily extended to store more variables.

When using serial SRAM to store variables, keep in mind that exception events, such as interrupts, could result in the register being saved to the stack if the associated function is being executed. This will have only a minor impact on internal SRAM usage in most applications.

Example Source Code

```
//*****************************************************************************
//
// serial_sram.c - Example for accessing an external serial SRAM device -
//                 AMI Semiconductor's 256Kb N256S0818HDA
//                 Runs on EKK-LM3S811 Eval kit
//
// Copyright (c) 2006 Luminary Micro, Inc. All rights reserved.
//
// Software License Agreement
//
// Luminary Micro, Inc. (LMI) is supplying this software for use solely and
// exclusively on LMI's Stellaris Family of microcontroller products.
//
// The software is owned by LMI and/or its suppliers, and is protected under
// applicable copyright laws. All rights are reserved. Any use in violation
// of the foregoing restrictions may subject the user to criminal sanctions
// under applicable laws, as well as to civil liability for the breach of the
// terms and conditions of this license.
//
// THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS, IMPLIED
// OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
// LMI SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR
// CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
//
//*****************************************************************************

#include "hw_memmap.h"
#include "hw_types.h"
#include "gpio.h"
#include "sysctl.h"
#include "ssi.h"
#include "osram96x16.h"

//*****************************************************************************
//
// This code writes an incremental value to every location in serial SRAM then
// verifies the write operation.  Byte-wide burst mode is used.  This allows
// the entire memory contents to be written/read in a single operation (if
// required).
//
//*****************************************************************************

//*****************************************************************************
//
// The GPIO port A pin numbers for the various SSI signals.
//
//*****************************************************************************
#define SSI_CS                  GPIO_PIN_3
#define SSI_CLK                 GPIO_PIN_2
```

```
#define SSI_TX                  GPIO_PIN_5
#define SSI_RX                  GPIO_PIN_4

//*****************************************************************************
//
// Commands that can be sent to the AMI N256S08018HDA
//
//*****************************************************************************
#define AMI_RAM_RD              0x03        // Read data memory
#define AMI_RAM_WR              0x02        // Write data memory
#define AMI_RAM_RDSR            0x05        // Read status register
#define AMI_RAM_WRSR            0x01        // Write status register

//*****************************************************************************
//
// Configuration options that can be written to the status register
//
//*****************************************************************************
#define MODE_WORD                0x00        // Word mode (default)
#define MODE_PAGE                0x80        // Page mode
#define MODE_BURST               0x40        // Burst mode
#define MODE_X8                  0x00        // 8 bit access (default)
#define MODE_X16                 0x20        // 16 bit access

//*****************************************************************************
//
// Other AMI SRAM defines
//
//*****************************************************************************
#define TEST_COUNTER_ADDRESS     0x7fff      // Store test counter here
#define AMI_DUMMY_WRITE          0           // Used to read-in data

//*****************************************************************************
//
// Global variables
//
//*****************************************************************************
char g_pcText[17];                           // Use to compile a line of text

//*****************************************************************************
//
// SramInit puts the SRAM device in burst mode then polls the status register
// for a valid status response.  Returns True if successful.
//
//*****************************************************************************
int
SramInit(void)
{
    unsigned long ulStatus;
    unsigned long ulDummyRead;

    //
    // Assert the SSI chip select.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, 0);

    //
    // Send the 'write status' instruction
    //
    SSIDataPut(SSI_BASE, AMI_RAM_WRSR);
    SSIDataGet(SSI_BASE, &ulDummyRead);

    //
    // Send the status register data
    //
    SSIDataPut(SSI_BASE, MODE_BURST + MODE_X8);
    SSIDataGet(SSI_BASE, &ulDummyRead);

    //
    // Release the SSI chip select to terminate the status register write op.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, SSI_CS);

    //
    // Assert the SSI chip select.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, 0);

    //
    // Send the 'read status' instruction
    //
    SSIDataPut(SSI_BASE, AMI_RAM_RDSR);
    SSIDataGet(SSI_BASE, &ulDummyRead);

    //
    // Get the status register data
    //
    SSIDataPut(SSI_BASE, AMI_DUMMY_WRITE);
    SSIDataGet(SSI_BASE, &ulStatus);

    //
    // Release the SSI chip select to terminate the status register read op.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, SSI_CS);

    //
    // Check Status and return
    //
    return ((ulStatus == MODE_BURST + MODE_X8) ? 1 : 0);

}

//*****************************************************************************
//
// SramWriteBurst
//
//*****************************************************************************
void
SramWriteBurst(unsigned long ulAddress, unsigned long ulLength,
            unsigned char *pucDataToWrite)
{
    unsigned long ulDummyRead;

    //
    // Assert the SSI chip select.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, 0);

    //
    // Send the write instruction
    //
    SSIDataPut(SSI_BASE, AMI_RAM_WR);
    SSIDataGet(SSI_BASE, &ulDummyRead);

    //
    // Send the 16-bit address as two bytes
    //
    SSIDataPut(SSI_BASE, (ulAddress >> 8) & 0xff);
    SSIDataGet(SSI_BASE, &ulDummyRead);
    SSIDataPut(SSI_BASE, ulAddress  & 0xff);
    SSIDataGet(SSI_BASE, &ulDummyRead);

    //
    // Now the data field
    //
    while (ulLength--)
    {
        SSIDataPut(SSI_BASE, *pucDataToWrite++);
        SSIDataGet(SSI_BASE, &ulDummyRead);
    }

    //
    // Release the SSI chip select.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, SSI_CS);

}

//*****************************************************************************
//
// SramReadBurst
//
//*****************************************************************************
void
SramReadBurst(unsigned long ulAddress, unsigned long ulLength,
            unsigned char *pucBuffer)
{
    unsigned long ulDummyRead;
    unsigned long ulReadData;

    //
    // Assert the SSI chip select.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, 0);

    //
    // Send the read instruction
    //
    SSIDataPut(SSI_BASE, AMI_RAM_RD);
    SSIDataGet(SSI_BASE, &ulDummyRead);
    //
    // Send the 16-bit address as two bytes
    //
    SSIDataPut(SSI_BASE, (ulAddress >> 8) & 0xff);
    SSIDataGet(SSI_BASE, &ulDummyRead);
    SSIDataPut(SSI_BASE, ulAddress & 0xff);
    SSIDataGet(SSI_BASE, &ulDummyRead);

    //
    // Now the data field
    //
    while (ulLength--)
    {
        //
        // Do a dummy write then get and store the data
        //
        SSIDataPut(SSI_BASE, AMI_DUMMY_WRITE);
        SSIDataGet(SSI_BASE, &ulReadData);
        *pucBuffer++ = (unsigned char)ulReadData;
    }

    //
    // Release the SSI chip select.
    //
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, SSI_CS);

}

//*****************************************************************************
//
// SramWrite writes a single byte to the specified address
//
//*****************************************************************************
void
SramWrite(unsigned long ulAddress, unsigned char ucDataToWrite)
{
    SramWriteBurst(ulAddress, 1, &ucDataToWrite);
}

//*****************************************************************************
//
// SramRead reads a single byte from the specified address
//
//*****************************************************************************
unsigned char
SramRead(unsigned long ulAddress)
{
    unsigned char ucData;

    SramReadBurst(ulAddress, 1, &ucData);
    return ucData;
}

//*****************************************************************************
//
// UpdateTestCounter keeps track of the number of times the SRAM test has
// run.  Normally the counter variable would be a static variable which uses
// one location in on-chip SRAM.  In this implementation, the counter is stored
// in serial SRAM to save a byte of on-chip SRAM and demonstrate how serial
// SRAM can be used for routine variable storage.
//
//*****************************************************************************
void
UpdateTestCounter(void)
{
    unsigned char ucTestCount;

    //
    // Read last counter value from serial SRAM and increment it
    //
    ucTestCount = SramRead(TEST_COUNTER_ADDRESS) + 1;

    //
    // Write counter value to OLED display
    //
    g_pcText[0] = '0' + ((ucTestCount / 100) % 10);
    g_pcText[1] = '0' + ((ucTestCount / 10) % 10);
    g_pcText[2] = '0' + (ucTestCount % 10);
    g_pcText[3] = '\0';
    OSRAMStringDraw(g_pcText, 78, 1);

    //
    // Store counter value for next time
    //
    SramWrite(TEST_COUNTER_ADDRESS, ucTestCount);
}


//*****************************************************************************
//
// This example writes then reads data to the serial SRAM
//
//*****************************************************************************
int
main(void)
{
    unsigned long ulAddress;
    unsigned long ulResult;

    //
    // Set the system clock to run at 50MHz
    //
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);

    //
    // Initialize the OLED display and display Title.
    //
    OSRAMInit(false);
    OSRAMStringDraw("32KB Serial SRAM", 0, 0);
    OSRAMStringDraw("Example", 0, 1);

    //
    // Enable the peripherals used by this example.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI);

    //
    // Configure the appropriate pins to be SSI instead of GPIO.  Note that
    // the chip select is kept as a GPIO to guarantee the appropriate
    // signalling to the AMI SRAM device.
    //
    GPIODirModeSet(GPIO_PORTA_BASE, SSI_CS, GPIO_DIR_MODE_OUT);
    GPIOPinWrite(GPIO_PORTA_BASE, SSI_CS, SSI_CS);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, SSI_CLK | SSI_TX | SSI_RX);

   //
    // Configure and enable the SSI port for master mode.
    // Operate at 4.17MHz - the fastest rate LM3S8xx devices can support
    //
    SSIConfig(SSI_BASE, SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER, 4167000, 8);
    SSIEnable(SSI_BASE);

    //
    // Loop forever writing and verifying data
    //
    while(1)
    {
        //
        // Initialize and confirm
        //
        ulResult = SramInit();

        //
        // Write test data
        //
        for (ulAddress=0; ulAddress < TEST_COUNTER_ADDRESS; ulAddress++)
        {
            SramWrite(ulAddress,(unsigned char)ulAddress);
        }

        //
        // Verify test data
        //
        for (ulAddress=0; ulAddress < TEST_COUNTER_ADDRESS; ulAddress++)
        {
            ulResult &= (SramRead(ulAddress)==(unsigned char)ulAddress) ? 1 : 0;
        }

        //
        // Display result on OLED display
        //
        if (ulResult)
        {
            OSRAMStringDraw("PASS", 48, 1);
        }
        else
        {
            OSRAMStringDraw("FAIL", 48, 1);
        }
        //
        // Update Test Counter on OLED display
        //
        UpdateTestCounter();

    }
}
```

Conclusion

As the example code shows, low-cost serial SRAM devices can effectively store standard static variables that would not otherwise fit in internal SRAM.

It is also now possible to use serial SRAMs with microcontrollers to store large data structures. Stellaris ARM Cortex-M3 based microcontrollers have significant processing capabilities useful in data acquisition and communications applications. A serial SRAM complements this capability by enabling data processing beyond the 8 KB internal SRAM.

References

The following are available for download at www.luminarymicro.com:

- Stellaris microcontroller data sheet, Publication Number DS-LM3S*nnn* (where *nnn* is the part number for that specific Stellaris family device)

In addition, the following document may be useful:

- AMI Semiconductor data sheet, *64Kb Low Power Serial SRAMs* (order number N64S0818HDA/ N64S0830HDA)

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless