AVR Freaks

DESIGN NOTE #028

AUTHOR: CHRIS R. MORSE, CHRIS@MORSETECH.NET
KEYWORDS: EXTERNAL SRAM SMALL AVR, XRAM

This document is originally distributed by AVRfreaks.net, and may be distributed, reproduced, and modified without restrictions. Updates and additional design notes can be found at: www.AVRfreaks.net

Using External SRAM with Small AVR Devices

Introduction

When using small AVR devices, internal SRAM is often limited and there are insufficient I/O pins to interface with an external memory (SRAM) device. This article examines a few ways one may overcome this limitation and provides a schematic/figure and code for one such method. The intention is not to provide a full working solution, although one is provided, but to provide a background of ideas to explore and allow the designer to choose from more options to create the solution that works best for the application at hand.

Background

Before discussing how to design an application to use external memory, it is helpful to examine how memory is typically used and accessed in embedded applications.

First, let us examine how memory is used. Most uses can be classified into three types of data: Operation, configuration, and storage. Operation data consists of items such as local variables, pointers, and the call stack. Configuration data consists of general parameters for operation and user or state defined values. This type of data is usually seen as blocks of fields (structures or “structs” in C) and is often used to initialize operation data. Storage is usually the input and/or output of the embedded system, for example: The conversion values of Analog to Digital Converters (ADC).

Second, we shall consider what type of memory addressing is necessary. Without hesitation, most designers would say direct random access is required. However, many embedded applications can be very effective, more efficient even, using other modes such as sequential addressing. Even those applications that truly require random access can often be reasonably accommodated with careful design.

The Challenge of Interfacing with External Memory

The challenge to using external memory comes down to basic economics-scarcity. Microcontrollers have a limited number of I/O pins and memory devices require an abundance of them, particularly for addressing. By using some of the address circuits described below it is possible to use external memory with small AVR devices such as the AT90S2313.

Traditionally, address lines are multiplexed using latches such as the 74HC573 family or sometimes a Shift Register. An alternate method is to use a Ripple Counter like the CD4040BC family. A Ripple Counter approach (described below) can address an unlim-

ited amount of SRAM using only two pins. Table 1 below compares these methods for I/O pin requirements and addressing speed.

Table 1. Comparison of External Memory Address Interfaces

Interface Method	I/O Pin Requirements	Address Access Speed
Direct	Very high	Very fast
Latch	High	Fast
Shift Register	Medium	Slow
Ripple Counter	Very low	Fast (sequential) Very slow (random)

Design Using Ripple Counters

Hardware

The hardware design for using Ripple Counters (also called Ripple Carry Binary Counters) is straightforward. The Ripple Counter requires a ripple input (AI) and a reset pin (AR). Both pins are used together by the application to control addressing. The AI pin increments the address (sequentially) and AR resets it to zero.

One can accommodate more address lines than available in a single Ripple Counter device by "daisy-chaining" the devices together. To do so, connect the most significant bit (MSB) of the first device to the input pin of the next device and so on for as many devices as necessary. Figure 1 below shows this circuit with U3 being incremented by U2, in effect; this adds an additional decade to the counter circuit. This entire circuit requires only 13 pins to address, control, and read/write data to the SRAM device. By comparison, using a Shift Register would require 17 pins and a typical latch circuit would require at least 21 pins and would be limited to addressing 16 lines (2^{16} = 64KB). For additional pin savings on either circuit a bi-directional Shift Register could be used for the data pins however, there is much more overhead in data access using this method.

Firmware/Software

No matter what circuit is used, the application should be designed to make optimal use of the addressing scheme employed. To do this successfully, the data must be partitioned. Operational data should almost always be stored in the fastest, most local memory, in this case, the AVR internal SRAM. Conversely, configuration and storage data can usually be moved to external memory. When using sequential addressing, items accessed frequently should be placed at the lowest addresses.

The example code below illustrates how to access the SRAM using the circuits shown in Figure 1. In data-logging type applications this is a very effective solution that provides fast access time and leaves enough device pins to accomplish the purpose of the application.

Enhancements

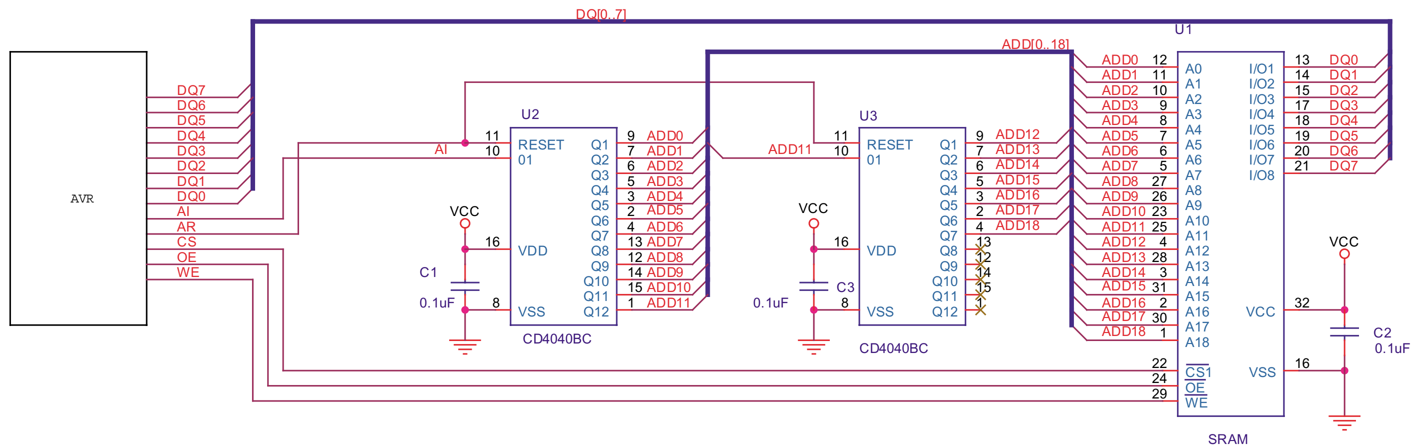
An enhancement for this scheme could be to make the most significant one or two address lines separately controlled by dedicated I/O pins, effectively making banks or pages in the SRAM. Doing so would have two benefits:

1. Reduce the amount of time to access higher addresses and
2. Reduce the amount of code required by using a smaller variable size for the tRamSize type.

Summary

This article has illustrated a few ways that AVR microcontrollers can interface with external SRAM using very few I/O pins and not sacrifice much functionality or performance. Hopefully, it has provided some new ideas for interfacing external memory with small AVR devices.

Figure 1. Interface to External SRAM Using Ripple Counters for Address Lines



Example Code

```
/*****************************************
* File:     SRAMlib.c
* Author:   Chris Morse, chris@morsetech.net
* Purpose:  Interface to JDEC standard
*           SRAM chips
*****************************************/
#include <io-avr.h>
#include <inttypes.h>
#include <pgmspace.h>
#include <iomacros.h>

//Change this for smaller addresses
typedef unsigned long tRamAddr;

//Stores current address on counters
static tRamAddr glAddr;

//These define how the sram is wired
//  and can be in a project header
#define SRAM_DATA_DDR       DDRB
#define SRAM_DATA_OUT       PORTB
#define SRAM_DATA_IN        PINB
#define SRAM_SCS_PORT       PORTD
#define SRAM_SCS_PIN        2
#define SRAM_SOE_PORT       PORTD
```

```
#define SRAM_SOE_PIN        3
#define SRAM_SWE_PORT       PORTD
#define SRAM_SWE_PIN        4
#define SRAM_SAR_PORT       PORTD
#define SRAM_SAR_PIN        5
#define SRAM_SAI_PORT       PORTD
#define SRAM_SAI_PIN        6

//Macros for SRAM Control
   //Put chip in Write mode
   #define SRAM_SET_WRITE() { \
      sbi(SRAM_SOE_PORT, SRAM_SOE_PIN); \
      cbi(SRAM_SWE_PORT, SRAM_SWE_PIN); \
      SRAM_DATA_DDR = 0xFF; }

   //Commit Write to device
   #define SRAM_COMMIT_WRITE() { \
      sbi(SRAM_SWE_PORT, SRAM_SWE_PIN); \
      cbi(SRAM_SWE_PORT, SRAM_SWE_PIN); }

   //Set to Write mode and output
   #define SRAM_SET_READ() { \
      cbi(SRAM_SOE_PORT, SRAM_SOE_PIN); \
      sbi(SRAM_SWE_PORT, SRAM_SWE_PIN); \
      SRAM_DATA_DDR = 0x00; }

   //Make sure SRAM is in wake mode
   #define SRAM_SELECT() \
      cbi(SRAM_SCS_PORT, SRAM_SCS_PIN);

   //Put the SRAM in standby
   #define SRAM_DESELECT() \
      sbi(SRAM_SCS_PORT, SRAM_SCS_PIN);

   //Reset the address latch
   #define SRAM_ADDR_RESET()  { \
      sbi(SRAM_SAR_PORT, SRAM_SAR_PIN); \
      cbi(SRAM_SAI_PORT, SRAM_SAI_PIN); \
      cbi(SRAM_SAR_PORT, SRAM_SAR_PIN); }

   //Increment the address latch
   #define SRAM_ADDR_INCR()   { \
      sbi(SRAM_SAI_PORT, SRAM_SAI_PIN); \
      cbi(SRAM_SAI_PORT, SRAM_SAI_PIN); }

//Functions for SRAM Actions
void SRAMsetAddr(const tRamAddr clAddr)
{
```

```
    tRamAddr iIncUnits;

    //Determine how much to increment
    if(glAddr < clAddr) {
        SRAM_ADDR_RESET();
        iIncUnits = clAddr;
    } else {
        iIncUnits = clAddr - glAddr;
    }

    //Increment the adddress latch
    while(iIncUnits--) {
        SRAM_ADDR_INCR();
    }

    //Store the new adddress
    glAddr = clAddr;
}


void* SRAMreadBuf(void* pbBuf, const tRamAddr clAddr, uint16_t ciSize)
{
    uint8_t* pData = (uint8_t*)pbBuf;

    //Setup the chip
    SRAMsetAddr(clAddr);
    SRAM_SELECT();
    SRAM_SET_READ();

    //Read the data
    while(ciSize > 0) {
        //Copy byte to memory pointer
        *pData = SRAM_DATA_IN;
        SRAM_ADDR_INCR();

        //Increment pointer and decr counter
        pData++;
        ciSize--;
    }

    SRAM_DESELECT();
    return(pbBuf);
}

void SRAMwriteBuf(const tRamAddr clAddr, const void* const cpbBuf, uint16_t
ciSize)
{
    uint8_t* pData = (uint8_t*)cpbBuf;
```

```
    //Setup the chip
    SRAMsetAddr(clAddr);
    SRAM_SELECT();
    SRAM_SET_WRITE();

    //Write the data
    while(ciSize > 0) {
        //Write data and commit
        SRAM_DATA_OUT = *pData;
        SRAM_COMMIT_WRITE();
        //Increment the address & pointer
        SRAM_ADDR_INCR();
        pData++;
        //Decrement the byte counter
        ciSize--;
    }

    SRAM_DESELECT();
}
```