Micromint, Inc	**AN703**
	Micro64/128
Accessing the 36k of SRAM	12/3/04

Introduction: Micro64/128 has a total of 36k of SRAM. 4 k of SRAM is built into the processor an additional 32k of SRAM is available inside the Micro64/128 module but external to the processor. For discussion purposes we will call the 4k that resides in the processor as internal SRAM and the additional 32k of SRAM as external SRAM. The internal SRAM's address runs from 0000 Hex to 10FF Hex. The external SRAM runs from address 1100 Hex to 7FFF Hex. Figure 1 shows a picture of the SRAM memory map. Looking at this memory map leads you to belive that there is only a total of 32k of SRAM. Looks can be deceiving. There is 4k of external SRAM hidden under the internal SRAM. This application note demonstrates how to access all of the external memory.

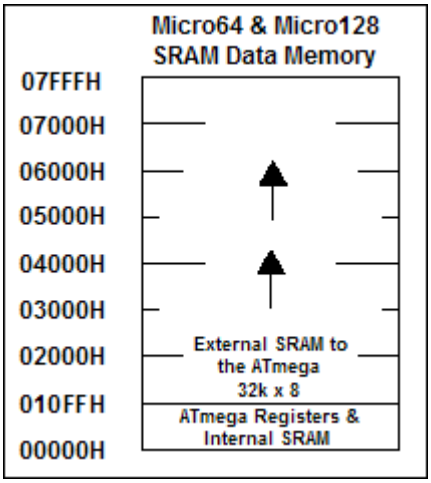
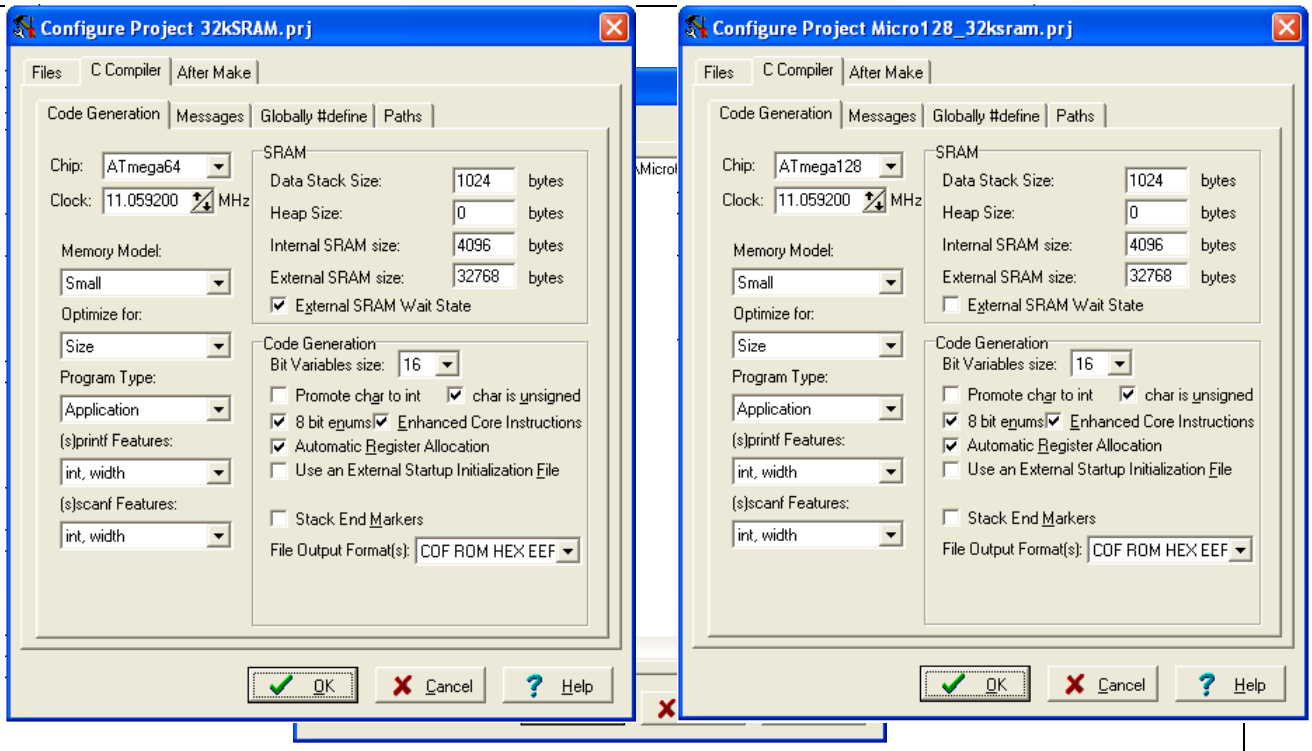


Figure 1: Micro64/128 SRAM Memory Map

Background: The SRAM for the Atmega64/128 can be set-up numerous ways. Please refer to the help topic SRAM Memory Organization for the CodeVision AVR C compiler for further information on how to use the memory allocation functions associated with the C programming language. This section will describe how to set-up the SRAM so that the compiler only uses the internal SRAM and the external SRAM will be used for the end user. It will also explain some of the problems that can ocurr by allocating the external space by hand. To set the SRAM up so you have full use of the external SRAM follow these steps:

1. Click on "Project".
2. Select "Configure". The following window should open



3. Click on “C Compiler” tab and the window should change something similar to the following:

Figure 3: Micro64 C Compiler Set up

Figure 4: Micro128 C Compiler Setup

4. Where the external SRAM says 32768 you want to change to 0 everything else in the above picture should remain the same depending on your application. If you wanted to let the compiler allocate the memory then you would not change the 32768 to 0 and you would not have to perform step 6.
5. Press the OK button.
6. In your program after you declare the local variables for main, you need to initialize the SRAM. You can do this by entering the following code:

```
 // External SRAM page configuration:
//            -          / 0000h - 7FFFh
MCUCR=0x80;
XMCRA=0x00;
```

The dangers of using SRAM this way:
When you compile the sample program you will receive a warning that will say “global variable out of range”. The compiler issues this warning because it is letting you know that it is not keeping track of the allocation of external memory. It is basically telling you try and write to a memory address that does not exist, like address 8000h, and it will not issue an error telling you about the mistake. If we left the 32768 in we would receive an error when trying to allocate any variables above 7FFFh.
This means that you need to know exactly how many bits a particular variable type takes up. Table 1 is a listing of all the variable types, their sizes, and their ranges. The way you would allocate the variable is as follows: unsigned char a @0x7fff; That would make allow any value from 0 to 255 that is assigned to “a” be stored in external memory address 7FFFh. Since an unsigned char only takes up a byte of memory this will work. If you were to allocate an int to 7FFFh the compiler wouldn’t issue an error but you would lose a byte of data because the external memory physically ends at 7FFFh. Please make sure you keep track of what addresses you assign variables to if you chose to do it this way. A good way to keep track would be to use a spreadsheet like Microsoft Excel.

Type	Size (Bits)	Range
bit	1	0 , 1
char	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
long int	32	-2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
signed long int	32	-2147483648 to 2147483647
float	32	±1.175e-38 to ±3.402e38
double	32	±1.175e-38 to ±3.402e38

Table 1: Data Types Supported by the Compiler, Range of Possible Values and Size in Bits

How it works: In order to access the external SRAM from address 1100h to 7FFFh you can assign a variable to a certain address or you can use a pointer to point to the address and move data into the SRAM address the pointer is pointing to. Keep in mind that if you allocate a char pointer it will only increment the address by a byte. An int will increment the address by two bytes. The example program demonstrates both of these techniques.
To access the first 4k of external SRAM the higher address bits must be masked to 0. This can be accomplished by setting PORTC to outputs, making them low, and then releasing PORTC.7, PORTC.6 and PORTC.5 from the high order address bus. XMM0 and XMM1 bits must be set in order to release PORTC.7, PORTC.6, and PORTC.5. After the write or read from SRAM is done simply clear the XXM0 and XMM1 bit. The sample program demonstrates how to read and write to external memory address 0x0000. The two routines are WRRAM_0000h() and RDRAM_0000h().

Program Listing:

```
/*****************************************
Program : 32kSRAM Example for Micro64
Company : Micromint, Inc.
*****************************************/

#include <mega64.h>
#include <MMRS485.h>              // Micromints Library for using both USARTs
#include <stdio.h>                // Standard I/O library
#include <delay.h>
// Macros to use in order to set individual bits in extended I/O space located in SRAM between
// 0x60 and 0xFF can not use the CBI or SBI assembly instruction to clear or set a bit.
```

```
#define SETBIT(port,bit)(port)|=1<<(bit)
#define CLRBIT(port,bit)(port)&=~(1<<(bit))
#define OFFSET 0x2000
// Declare your global variables here
int COM;                        // if COM = 0 then use USART0 if it = 1 then use USART1
int CH0 @0x7ffe;

unsigned int Data @0xFFE;           // This the memory location where the 12-bit ADC conversion
                            //count is stored after calling the utility to read a channel

// Routines used to call the utilities for the 12-bit ADC
void(*SingleADC1)(void)=0x7BFD;
void(*SingleADC0)(void)=0x7C04;
void RD_RAM(unsigned int);
void WR_RAM(unsigned int);
void WRRAM_0000h(unsigned int, unsigned char);
void RDRAM_0000h(unsigned int);
void main(void)
{
// Declare your local variables here

// External SRAM page configuration:
//             -             / 0000h - 7FFFh
// Lower page wait state(s): None
// Upper page wait state(s): None
MCUCR=0x80;
XMCRA=0x00;
CLRBIT(XMCRB,0);  //Make XMM0 bit 0 of the XMCRB register low enabling all of external SRAM
CLRBIT(XMCRB,1);  //Make XMM0 bit 1 of the XMCRB register low
CLRBIT(XMCRB,2);  //Make XMM0 bit 2 of the XMCRB register low

// Set up USART1's Baud rate at 9600 bps with a 11.0592 MHz Crystal
UCSR1A=0x00;      // RX EN, TX EN
UCSR1B=0x18;     // RX EN, TX EN
UCSR1C=0x06;     // 8N1
UBRR1H=0x00;    // Baud rate high - 9600
UBRR1L=0x47;   // Baud rate low

COM = 1;                        // Use USART1
DDRD.6 = 0;                             // Make PORTD.6 an output
PORTD.6 = 1;                            // Enable the RS485 control line

while (1)
     {
     (*SingleADC0)();
     CH0 = Data;
     printf("CH0 = %d\r\n",CH0);
     printf("\r\n");
     WR_RAM(0x2000);
     WRRAM_0000h(0x0000, 39);
     WRRAM_0000h(0x0001, 40);
     WRRAM_0000h(0x0002, 41);
     RDRAM_0000h(0x0000);
     RDRAM_0000h(0x0001);
     RDRAM_0000h(0x0002);
     RD_RAM(0x2000);
     delay_ms(3000);
     };
}

void WR_RAM(unsigned int MemAddr)
```

```
{
  int *p;
  p=MemAddr;
  while(p < (MemAddr + 10))
  {
   (*SingleADC1)();
   *p = Data;
   printf("Wrote %d", Data);
   printf(" to location %X", p);
   printf("    \r\n");
   p++;
  }
  printf("\r\n");
}

void RD_RAM(unsigned int MemAddr)
{
  int CH1;
  int *p;
  p=MemAddr;
  while(p < (MemAddr + 10))
  {
   CH1=*p;
   printf("Read %d", CH1);
   printf(" at location %X", p);
   printf("    \r\n");
   p++;
  }
  printf("\r\n");
}

void WRRAM_0000h(unsigned int MemAddr, unsigned char WRData)
{
  unsigned char *p = (unsigned char *) (OFFSET + 1);   // This assignment is called casting.
  p = OFFSET + MemAddr;

  DDRC = 0xFF;   // Set the released port pin PORTC.7, PORTC.6, and PORTC.5 to outputs
  PORTC = 0x00;  // Make the outputs low
  // Settng the following two bits releases some of PORTC from the External Memory Access
  SETBIT(XMCRB,0);  //Make XMM0 bit 0 and bit 1 of the XMCRB register high to make external
  SETBIT(XMCRB,1);  //SRAM enabled up to address 1FFFH.
  *p = WRData;
   printf("Wrote %d", WRData);
   printf(" to location %X", p-OFFSET);
   printf("    \r\n");
   printf("\r\n");
  // Clear the following two bits to make all of PORTC used for External Memory Access
  CLRBIT(XMCRB,0);  //Make XMM0 bit 0 of the XMCRB register low enabling all of external SRAM
  CLRBIT(XMCRB,1);  //Make XMM0 bit 1 of the XMCRB register low
}

void RDRAM_0000h(unsigned int MemAddr)
{
 unsigned char *p = (unsigned char *) (OFFSET + 1);   // This assignment is called casting.
 unsigned char RDData;
   p = OFFSET + MemAddr;
   DDRC = 0xFF;   // Set the released port pin PORTC.7, PORTC.6, and PORTC.5 to outputs
   PORTC = 0x00;  // Make the outputs low
   // Settng the following two bits releases some of PORTC from the External Memory Access
   SETBIT(XMCRB,0);  //Make XMM0 bit 0 and bit 1 of the XMCRB register high to make external
   SETBIT(XMCRB,1);  //SRAM enabled up to address 1FFFH.
```

```
    RDData = *p;
    printf("Read %d", RDData);
    printf(" to location %X", p-OFFSET);
    printf("    \r\n");
    printf("\r\n");
   // Clear the following two bits to make all of PORTC used for External Memory Access
    CLRBIT(XMCRB,0);  //Make XMM0 bit 0 of the XMCRB register low enabling all of external SRAM
    CLRBIT(XMCRB,1);  //Make XMM0 bit 1 of the XMCRB register low
}
```