AVR Freaks

DESIGN NOTE #007

KEYWORDS:
C, LITTLE ENDIAN, BIG ENDIAN, COMMUNICATION PROBLEMS

Little and Big Endian

Introduction

This document describes how to handle the terms Little and Big Endian, which can cause confusion when using some communication protocols.

Overview

The terms Little and Big Endian indicate the byte order of 16- or 32-bit words when the data is sent between communicating units or stored in a memory.

When communicating, Little Endian means that the LSB (Least Significant Byte) is sent first followed by the higher order bytes. In a memory, Little Endian means that the LSB is stored at the lowest storage address.

When communicating, Big Endian means that the MSB (Most Significant Byte) is sent first followed by the lower order bytes. In a memory, Big Endian means that the MSB is stored at the lowest storage address. Big Endian is also called network byte order.

Data stored in the AVR program and data memories, are normally accessed as Little Endian (Even if the Flash program memory is physically organized as Big Endian). The AVR Register file architecture allows handling and switching between Little and Big Endian without any code overhead.

The following example demonstrates how an incoming data stream buffer can be swapped from Big Endian to Little Endian and conversely. The swapping takes place in the Register file while data is stored in the AVR Data Memory.

```
// Useful macros for extracting high and low byte of 16-bit words
#define LO(x) ((unsigned char) ((x) & 0x00ff))     // Low byte of an integer
#define HI(x) ((unsigned char) (((x) >> 8) & 0x00ff)) // High byte of an
integer

unsigned int *preceive = receiveData;           // Incoming data Buffer
unsigned int *pstore = storeData;               // Data memory pointer
unsigned char length;                           // Size of incoming buffer
unsigned int temp;

length = RECEIVE_DATA_LENGTH;                   // Assign length to variable
do
{
    temp = *preceive++;                       // Read word into register file
    *pstore++ = ((LO(temp)<<8) | (HI(temp)));   // Store in reverse order
} while (--length);                           // Continue until end of buffer
```

The temp variable is used to generate optimized code for some compilers.

This method can also be used for copying data from memory-mapped data to SRAM data memory.