App Note 3477: Using the Free SDCC C Compiler to Develop Firmware for the DS89C420/430/440/450 Family of Microcontrollers

The SDCC (Small Devices C Compiler) is a free C compiler developed for 8-bit microcontrollers. This application note demonstrates how to use the SDCC to develop firmware for the DS89C420/430/440/450 family of ultra-high-speed 8051-compatible microcontrollers. Installing the SDCC free C compiler is also explained.

Introduction

The SDCC (Small Devices C Compiler) is a free C compiler developed for 8-bit microcontrollers. Although compatible with many different architectures, the SDCC compiler has extended support for devices based on the 8051-core. This application note will focus on using the SDCC to develop firmware for the DS89C420/430/440/450 family of ultra-high-speed 8051-compatible microcontrollers from Maxim/Dallas Semiconductor.

The SDCC is a command line, firmware development tool that includes a preprocessor, a compiler, an assembler, a linker, and an optimizer. Also bundled with the install file is the SDCDB, a source level debugger similar to gdb (GNU Debugger). When an error-free program is compiled and linked with the SDCC, a Load Module in Intel hex format is created. This file can then be loaded into the DS89C420/430/440/450 microcontroller's flash memory using a Serial Loader. (See DS89C420/430/440/450 documentation and application notes for details on downloading firmware to device.

For the most up-to-date information about the SDCC, visit http://sdcc.sourceforge.net or read the SDCC manual, sdccman.pdf (copied to your hard drive during installation). Questions can also be submitted to the online SDCC message forum or mailing list which can be found in the "Support" section of the SDCC webpage.

Installing the SDCC Free C Compiler

To install the SDCC, download the latest version from the "Download" section of the SDCC website at http://sdcc.sourceforge.net. Although nightly builds of the software are available, it is usually safest to download the latest fully tested release version (currently v2.4.0).

On the "Download" page, builds of the SDCC are available for several different operating systems. If you are working on a PC running Microsoft Windows, download the win32 self-executing SDCC install file (sdcc-2.4.0-setup.exe) and run the executable.

When installing the program, a prompt will appear asking to add the directory containing the program

binaries to your path. This is recommended, and the remainder of this application note will assume that the user has done so.

Compiling a Simple C Program with the SDCC Compiler

To ensure that the SDCC installed correctly on your hard drive, open a Command Prompt and type sdcc --version. Press [Enter], and the text displayed in **Figure 1** should appear in the window (actual text will depend on the SDCC version that you downloaded):

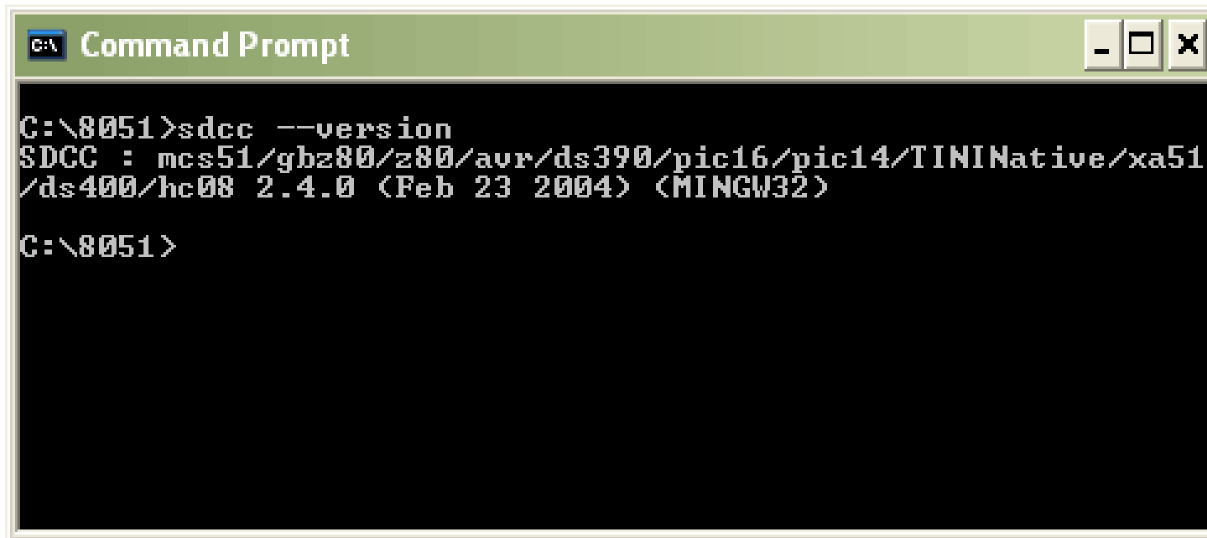

Figure 1. Verifying the Correct Installation of the SDCC by Performing a Version Check

To test the include path, create a file called sdcctest.c and copy the following source code into the file.

```
#include <string.h>

char str[6] = "MAXIM";
bit  flag;

void main(void)
{
     if (strcmp(str,"MAXIM") == 0)
          flag = 0;
     else
          flag = 1;

     while(1); // program loop
}
```

Save the file in plain ASCII format (i.e., using Microsoft Notepad). In the Command Prompt, type sdcc sdcctest.c and press [Enter]. If nothing appears, as shown in **Figure 2**, the program compiled successfully.

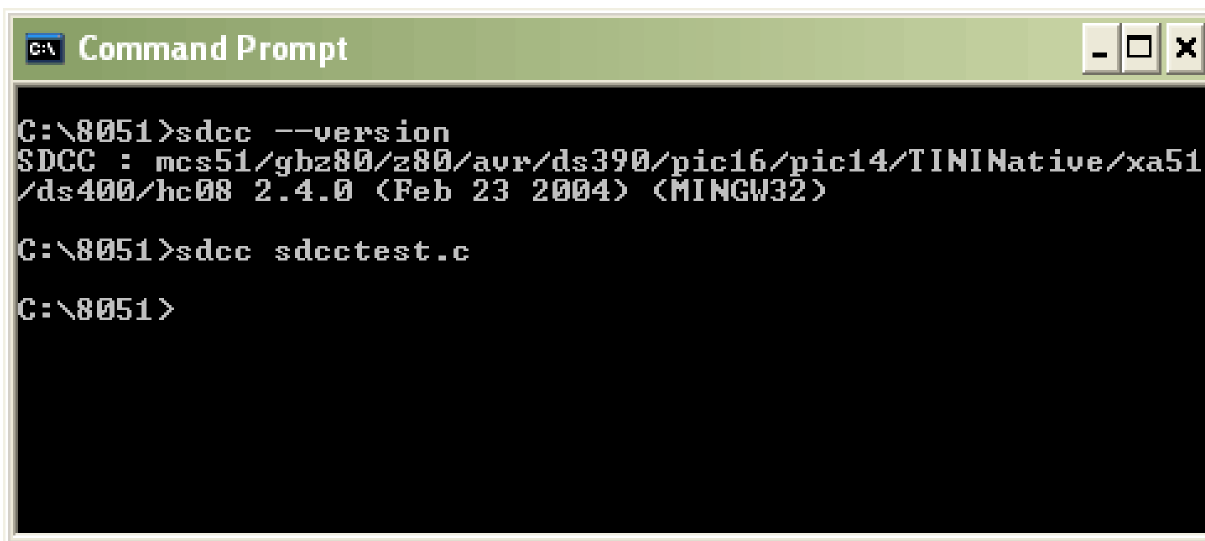

Figure 2. Compiling a Simple SDCC Program

Many files are created by the SDCC when you compile your source code successfully. In the directory to which you compiled, you will find the following files:

- sdcctest.asm: the assembler file for your program
- sdcctest.lst: the listing file for your program
- sdcctest.rst: the listing file for your program updated by the linker
- sdcctest.map: the final memory map updated by the linker
- sdcctest.ihx: the Load Module in Intel hex format. This file must be downloaded into the microcontroller.

Other files are also created (many of them for the source level debuggers). Read the SDCC documentation for further information.

Data Types Specific to the SDCC

The SDCC supports most ANSI-C data types such as:

- char: 1 byte - can be signed or unsigned
- short: 2 bytes - can be signed or unsigned
- int: 2 bytes - can be signed or unsigned
- long: 4 bytes - can be signed or unsigned
- float: 4 bytes

In addition, the SDCC supports a number of extended data types (also called storage classes) to take advantage of the 8051-architecture. They are presented in the following subsections with examples.

Unlike some commercial 8051 microcontroller development tools, the SDCC is only capable of declaring Special Function Registers as both bit and byte addressable. Although supported by the 8051 assembly language, shared bit and byte addressable RAM is not supported by the SDCC. To prove this, observe the following code sample and compiled assembler code.

C source code:

```
union
{
  unsigned char a_byte;
  struct
  {
    unsigned char bit0 : 1;
    unsigned char bit1 : 1;
    unsigned char bit2 : 1;
    unsigned char bit3 : 1;
    unsigned char bit4 : 1;
    unsigned char bit5 : 1;
    unsigned char bit6 : 1;
    unsigned char bit7 : 1;
  } a_bit;
} a;

bit b;

void main(void)
{
     a.a_byte     = 0x05;
     a.a_bit.bit6 = 1;
     b            = 1;

     while(1); // program loop
}
```

Assembly listing (.rst file):

```
                            ...
                            159 ;sdcctest.c:21: a.a_byte = 5;
                            160 ;     genPointerSet
                            161 ;     genNearPointerSet
                            162 ;     genDataPointerSet
   0031 75 21 05            163         mov     _a,#0x05
                            164 ;sdcctest.c:23: a.a_bit.bit6 = 1;
                            165 ;     genPointerSet
                            166 ;     genNearPointerSet
   0034 78 21               167         mov     r0,#_a
                            168 ;     genPackBits
   0036 E6                  169         mov     a,@r0
   0037 44 40               170         orl     a,#0x40
   0039 F6                  171         mov     @r0,a
                            172 ;sdcctest.c:25: b = 1;
                            173 ;     genAssign
   003A D2 00               174         setb    _b
                            175 ;sdcctest.c:27: while(1); // program loop
                            ...
```

Although the bit fields in declaration of "a" appear to be bit-addressable memory, the assembly listing (taken from the .rst file generated by SDCC) shows that the variable does not use bit addressing. In the listing do not confuse "a" and "_a". The "a" refers to the accumulator while the "_a" refers to the variable.

Note that the "Absolute Addressing" section of this appnote presents an interesting way to allocate memory so it acts as true bit-addressable memory.

near/data

Declaring a variable with the **near** or **data** storage class places the variable in directly addressable RAM in the 8051-core. The DS89C420/430/440/450 family of microcontrollers has 128 bytes of directly addressable memory available. This is the fastest type of memory that can be accessed by the 8051, and the assembly code generated to read or write data in this area of RAM requires a single MOV instruction.

```
#include "sdcc_reg420.h"

data unsigned char outPort0 = 0x4A;

void main(void)
{
     P0 = outPort0;

     while (1); // program loop
}
```

The sdcc_reg420.h definition file used in this example is given in Appendix A.

far/xdata

Declaring a variable with the **far** or **xdata** storage class places the variable in external RAM. Although this gives the developer access to a much larger RAM space, the assembly code generated to read and write to this memory uses a MOVX instruction which requires loading the external memory address into the data pointer.

The DS89C420/430/440/450 family of microcontrollers includes a 1 kilobyte internal SRAM that can be used for variables declared with **far/xdata**. Note that the DME1:0 bits in the Power Management Register (PMR) must be set for internal SRAM mode before this memory can be initialized or used.

```
#include "sdcc_reg420.h"

xdata unsigned char ioPorts[2];

void main(void)
{
     PMR  |= 0x01; // Enable internal 1K SRAM

     ioPorts[0] = 0x4A;
     ioPorts[1] = 0x56;
```

```
     P0          = ioPorts[0];
     P1          = ioPorts[1];

     while (1); // program loop
}
```

idata
Declaring a variable with the **idata** storage class places the variable in indirectly addressable memory within the 8051-core. Indirectly addressable memory is similar to directly addressable memory as there are 128-bytes available in the 8051-core (not including Special Function Registers). Accessing **idata**, however, requires an extra MOV command to move the RAM address into a working register.

```
#include "sdcc_reg420.h"

idata unsigned int port0_x2;

void main(void)
{
     while (1) // program loop
     {
          port0_x2 = P0 * 2;
     }
}
```

pdata
The pdata storage class is used to access paged external data memory. This memory type is beyond the scope of this application note and interested readers are encouraged to review the pdata section in the SDCC documentation.

code
Declaring a variable with the **code** storage class indicates that the variable will be placed in program memory (specifically flash memory within the DS89C420/430/440/450 microcontrollers). The variables are read-only for the SDCC, therefore use **code** to declare constants (such as lookup tables) in your program.

```
#include "sdcc_reg420.h"

code unsigned char out[10] = {0x03,0x45,0xFA,0x43,0xDD,
                              0x1A,0xE0,0x00,0x87,0x91};

void main(void)
{
     data unsigned char i = 0;

     while (1) // program loop
     {
          P0 = out[i++];
          if (i==10)
```

```
            i=0;
      }
}
```

bit

Declaring a variable with the **bit** storage class places it in bit-addressable memory in the 8051-core. The 8051-core has 16 bytes of direct addressable RAM that can act as bit-addressable memory (bytes 0x20 to 0x2F), providing 128 total addressable bits. Variables of this type are efficient use of memory for flags.

```
#include "sdcc_reg420.h"

#define ESCAPE 0x1B

bit esc_char_flag = 0;

void main(void)
{
     P1 = 0x00;

     while (!esc_char_flag)
     {
          if (P0 == ESCAPE)
               esc_char_flag = 1;
     }

     P1 = 0xFF;

     while (1); // program loop
}
```

sfr

The **sfr** storage class defines a specific Special Function Register (SFR) in the 8051-core. The definition file presented in Appendix A defines all the SFRs in the DS89C420/430/440/450 microcontrollers using the **sfr** identifier.

Note that the following example defines the SFRs. Including the definition file sdcc_reg420.h was therefore not necessary.

```
sfr at 0x80 P0;
sfr at 0x90 P1;

void main(void)
{
     P0 = 0x00;
     P1 = 0xFF;

     while (1); // program loop
}
```

sbit

The **sbit** storage class defines a specific bit inside a bit-addressable SFR. In the 8051-core, all SFRs with an address that finishes with either a 0 or an 8 (in hex) are bit addressable. The definition file presented in Appendix A defines all the SFR addressable bits in the DS89C420/430/440/450 microcontroller using the **sbit** identifier.

```
sfr  at 0x80 P0;   // Port 0

sbit at 0x80 P0_0; // Port 0 bit 0
sbit at 0x81 P0_1; // Port 0 bit 1
sbit at 0x82 P0_2; // Port 0 bit 2
sbit at 0x83 P0_3; // Port 0 bit 3
sbit at 0x84 P0_4; // Port 0 bit 4
sbit at 0x85 P0_5; // Port 0 bit 5
sbit at 0x86 P0_6; // Port 0 bit 6
sbit at 0x87 P0_7; // Port 0 bit 7

void main(void)
{
     P0   = 0x00; // P0 = 0x00

     P0_4 =    1; // P0 = 0x10

     while (1);   // program loop
}
```

Absolute Addressing

Absolute addressing is supported by the SDCC using the **at** identifier. The SDCC will not, unfortunately, track variables declared at absolute addresses and may declare other variables so that they will overlap.

The following example program presents interesting potential bugs.

```
#include "sdcc_reg420.h"

unsigned char           a     = 0x4A;
unsigned int            b     = 0x0000;
unsigned char           c[64] = {0x00};

unsigned char at 0x0010 y;
unsigned char at 0x0010 z;

void main(void)
{
     for(b=0; b<64; b++)
          c[b] = 0xAA;

     y = 0xF1;
     z = 0xF2;
```

```
    a = c[5];

    while (1); // program loop
}
```

Using the SDCC, the example compiles without any errors or any warnings, even though two variables, "**y**" and "**z**", are assigned to the same location. Next, if we were to run this program, we would expert the final assignment in the program (**a = c[5]**) to set "**a**" to 0xAA. But this is not the case. The actual value of "**a**" after the final assignment is 0xF2.

We can see the reason for this strange result by examining the following lines in the .map file created by SDCC that show the actual addresses used for each variable.

```
Area                                Addr   Size   Decimal Bytes (Attributes)
--------------------------------    ----   ----   ------- ----- ------------
.  .ABS.                            0000   0000 =      0. bytes (ABS,OVR)

      Value  Global
   --------  --------------------------------
     ...
     0010    _y
     0010    _z
     ...

Area                                Addr   Size   Decimal Bytes (Attributes)
--------------------------------    ----   ----   ------- ----- ------------
DSEG                                0008   0043 =     67. bytes (REL,CON)

      Value  Global
   --------  --------------------------------
     0008    _a
     0009    _b
     000B    _c
```

Note that the underscore placed at the beginning of the variable name is added by the compiler. If "**c**" will be located at address 0x000B and will be an array 64 bytes long, it will overlap the "**y**" and "**z**" variables at address 0x0010.

A use for absolute addressing is to simulate bit-addressable variables. In the following example, we choose to define the variable **n_byte** at the last byte location in bit-addressable memory. Next, we define **n_bit0** to **n_bit7** in the last 8 bit locations of bit-addressable memory in the 8051-core. As these overlap, the **n_byte** variable can be bit addressed using the **n_bit0** to **n_bit7** variables.

```
#include "sdcc_reg420.h"

data unsigned char at 0x002F n_byte;

bit                at 0x78   n_bit0;
```

```
bit                 at 0x79   n_bit1;
bit                 at 0x7A   n_bit2;
bit                 at 0x7B   n_bit3;
bit                 at 0x7C   n_bit4;
bit                 at 0x7D   n_bit5;
bit                 at 0x7E   n_bit6;
bit                 at 0x7F   n_bit7;

void main(void)
{
     n_byte = 0x00;
     n_bit4 = 1;

     P0     = n_byte; // P0 = 0x10

     while (1);       // program loop
}
```

Memory Models

The SDCC supports two memory models: small and large. When using the small memory model, the SDCC declares all variables without a storage class (i.e., **data, idata, xdata, pdata, bit, code**) in internal RAM. When using the large memory model, the SDCC declares all variables without a storage class in external RAM.

The small memory model is the default when compiling with the SDCC. To force the SDCC to use a particular memory model, add the following command line parameters:

```
sdcc --model-small sdcctest.c
```

or

```
sdcc --model-large sdcctest.c
```

Never combine modules or objects files compiled with different storage classes.

Interrupts in SDCC

The following format should be used to define an Interrupt Service Routine (ISR):

```
void interrupt_identifier (void) interrupt interrupt_number using bank_number
{
     ...
}
```

The ***interrupt_identifier*** can be any valid SDCC function name. The ***interrupt_number*** is the interrupt's position within the interrupt vector table. **Table 1** shows the Interrupt Numbers for every interrupt supported by the DS89C420/430/440/450 family of microcontrollers. The ***bank_number*** is an

optional parameter that indicates to the SCDD which register bank to use for storing local variables in the ISR.

Table 1. Interrupt Numbers for DS89C420/430/440/450 Interrupts Service Routines

Interrupt Name	Interrupt Vector	Interrupt Number
External Interrupt 0	0x03	0
Timer 0 Overflow	0x0B	1
External Interrupt 1	0x13	2
Timer 1 Overflow	0x1B	3
Serial Port 0	0x23	4
Timer 2 Overflow	0x2B	5
Power Fail	0x33	6
Serial Port 1	0x3B	7
External Interrupt 2	0x43	8
External Interrupt 3	0x4B	9
External Interrupt 4	0x53	10
External Interrupt 5	0x5B	11
Watchdog Interrupt	0x63	12

The SDCC handles many details involved with programming ISRs, like saving and restoring accumulators and the data pointer to and from the stack. (This is actually done for all functions as well. Refer to the **_naked** keyword in the SDCC manual to disable saving these variables to the stack.) Other details are not handled by the SDCC (for good reason) and present traps to developers new to embedded programming. Many of these issues are advanced programming concepts beyond the scope of this article, but the SDCC manual as well as embedded programming textbooks can provide more insight.

The following rules should be followed when using interrupts.

- Every global variable that can be written to inside an ISR and that is accessed outside the ISR must be declared as **volatile**. This will ensure that the optimizer does not remove instructions relating to this variable.
- Disable interrupts whenever using data in a non-atomic way (i.e., accessing 16-bit/32-bit variables). Access to a variable is atomic when the processor cannot interrupt (with an ISR) storing and loading data to and from memory.
- Avoid calling functions from within an ISR. If you must do this, declare the function as **reentrant** (see SDCC manual) which allocates all local variables in the function on the stack instead of in RAM.

Note that if your SDCC program uses ISRs located in source files other than the one that contains your **main()** function, the source file containing the **main()** function should include a function prototype for each of these ISRs.

The example below defines an ISR to handle Serial Communication Interface 1 (SCI_1) Interrupt Service Routines. The program receives a byte from the SCI_1 receiver, increments the byte by one, and transmits it continuously through the SCI_1 transmitter.

```
#include "sdcc_reg420.h"

volatile unsigned char n = 0x4A;

void sci1ISR (void) interrupt 7
{
    if (RI_1)
    {
        n     = SBUF1+1;     // Save Rx byte
        RI_1  = 0;           // Reset SCI_1 Rx interrupt flag
    }
    else if (TI_1)
    {
        SBUF1 = n;           // Load byte to Tx
        TI_1  = 0;           // Reset SCI_1 Tx interrupt flag
    }
}

void main(void)
{
    // 1. Init Serial Port
    EA    = 0;              // Enable global interrupt mask
    SCON1 = 0x50;           // Set SCI_1 to 8N1, Rx enabled
    TMOD |= 0x20;           // Set Timer 1 as Mode 2
    TH1   = 0xDD;           // Set SCI_1 for 2400 baud
    TR1   = 1;              // Enable Timer 1
    ES1   = 1;              // Enable interrupts for SCI_1
    EA    = 1;              // Disable global interrupt mask

    // 2. Initiate SCI_1 Tx
    SBUF1 = n;

    // 3. Program loop...
    while (1);
}
```

Inline Assembly

Inline assembly is fully supported by the SDCC. To use this feature, enclose the assembly code between the **_asm** and **_endasm** identifiers. Note that inline assembly code can also access C variables by prefixing the variable name with an underscore character. The following example uses inline assembly to perform **nop** instructions (used to waste a clock cycle within the microcontroller) and then increment the variable "**a**" by one.

```
#include "sdcc_reg420.h"
```

```
unsigned char a;

void main(void)
{
    // program loop...
    while (1)
    {

          a = P0;

          _asm
               nop
               nop
               nop
               inc _a
          _endasm;

          P1 = a;
    }
}
```

The SDCC is also capable of interfacing C and assembly functions. This is an advanced topic; refer to the SDCC manual for details.

APPENDIX A: SFR Definitions File for the DS89C420/430/440/450 (sdcc_reg420.h)

```
/*
 * sdcc_reg420.h
 *
 * Author: Paul Holden
 * MAXIM INTEGRATED PRODUCTS
 *
 * Special Function Register definitions file
 * DS89C420/430/440/450 Ultra-High Speed 8051-compatible uCs
 *
 */

#ifndef __REG420_H__
#define __REG420_H__

/*  BYTE Registers  */
sfr at 0x80 P0;
sfr at 0x81 SP;
sfr at 0x82 DPL;
sfr at 0x83 DPH;
sfr at 0x84 DPL1;
sfr at 0x85 DPH1;
sfr at 0x86 DPS;
```

```
sfr at 0x87 PCON;
sfr at 0x88 TCON;
sfr at 0x89 TMOD;
sfr at 0x8A TL0;
sfr at 0x8B TL1;
sfr at 0x8C TH0;
sfr at 0x8D TH1;
sfr at 0x8E CKCON;
sfr at 0x90 P1;
sfr at 0x91 EXIF;
sfr at 0x96 CKMOD;
sfr at 0x98 SCON0;
sfr at 0x99 SBUF0;
sfr at 0x9D ACON;
sfr at 0xA0 P2;
sfr at 0xA8 IE;
sfr at 0xA9 SADDR0;
sfr at 0xAA SADDR1;
sfr at 0xB0 P3;
sfr at 0xB1 IP1;
sfr at 0xB8 IP0;
sfr at 0xB9 SADEN0;
sfr at 0xBA SADEN1;
sfr at 0xC0 SCON1;
sfr at 0xC1 SBUF1;
sfr at 0xC2 ROMSIZE;
sfr at 0xC4 PMR;
sfr at 0xC5 STATUS;
sfr at 0xC7 TA;
sfr at 0xC8 T2CON;
sfr at 0xC9 T2MOD;
sfr at 0xCA RCAP2L;
sfr at 0xCB RCAP2H;
sfr at 0xCC TL2;
sfr at 0xCD TH2;
sfr at 0xD0 PSW;
sfr at 0xD5 FCNTL;
sfr at 0xD6 FDATA;
sfr at 0xD8 WDCON;
sfr at 0xE0 ACC;
sfr at 0xE8 EIE;
sfr at 0xF0 B;
sfr at 0xF1 EIP1;
sfr at 0xF8 EIP0;

/* BIT Registers  */

/* P0 */
sbit at 0x80 P0_0;
```

```
sbit at 0x81 P0_1;
sbit at 0x82 P0_2;
sbit at 0x83 P0_3;
sbit at 0x84 P0_4;
sbit at 0x85 P0_5;
sbit at 0x86 P0_6;
sbit at 0x87 P0_7;

/* TCON */
sbit at 0x88 IT0;
sbit at 0x89 IE0;
sbit at 0x8A IT1;
sbit at 0x8B IE1;
sbit at 0x8C TR0;
sbit at 0x8D TF0;
sbit at 0x8E TR1;
sbit at 0x8F TF1;

/* P1 */
sbit at 0x90 P1_0;
sbit at 0x91 P1_1;
sbit at 0x92 P1_2;
sbit at 0x93 P1_3;
sbit at 0x94 P1_4;
sbit at 0x95 P1_5;
sbit at 0x96 P1_6;
sbit at 0x97 P1_7;

/* SCON0 */
sbit at 0x98 RI_0;
sbit at 0x99 TI_0;
sbit at 0x9A RB8_0;
sbit at 0x9B TB8_0;
sbit at 0x9C REN_0;
sbit at 0x9D SM2_0;
sbit at 0x9E SM1_0;
sbit at 0x9F SM0_0;
sbit at 0x9F FE_0;

/* P2 */
sbit at 0xA0 P2_0;
sbit at 0xA1 P2_1;
sbit at 0xA2 P2_2;
sbit at 0xA3 P2_3;
sbit at 0xA4 P2_4;
sbit at 0xA5 P2_5;
sbit at 0xA6 P2_6;
sbit at 0xA7 P2_7;
```

```
/* IE */
sbit at 0xA8 EX0;
sbit at 0xA9 ET0;
sbit at 0xAA EX1;
sbit at 0xAB ET1;
sbit at 0xAC ES0;
sbit at 0xAD ET2;
sbit at 0xAE ES1;
sbit at 0xAF EA;

/* P3 */
sbit at 0xB0 P3_0;
sbit at 0xB1 P3_1;
sbit at 0xB2 P3_2;
sbit at 0xB3 P3_3;
sbit at 0xB4 P3_4;
sbit at 0xB5 P3_5;
sbit at 0xB6 P3_6;
sbit at 0xB7 P3_7;

/* IP0 */
sbit at 0xB8 LPX0;
sbit at 0xB9 LPT0;
sbit at 0xBA LPX1;
sbit at 0xBB LPT1;
sbit at 0xBC LPS0;
sbit at 0xBD LPT2;
sbit at 0xBE LPS1;

/* SCON1 */
sbit at 0xC0 RI_1;
sbit at 0xC1 TI_1;
sbit at 0xC2 RB8_1;
sbit at 0xC3 TB8_1;
sbit at 0xC4 REN_1;
sbit at 0xC5 SM2_1;
sbit at 0xC6 SM1_1;
sbit at 0xC7 SM0_1;

/* T2CON */
sbit at 0xC8 CP_RL_2;
sbit at 0xC9 C_T_2;
sbit at 0xCA TR_2;
sbit at 0xCB EXEN_2;
sbit at 0xCC TCLK;
sbit at 0xCD RCLK;
sbit at 0xCE EXF_2;
sbit at 0xCF TF_2;
```

```
/* PSW */
sbit at 0xD0 PARITY;
sbit at 0xD0 P;
sbit at 0xD1 F1;
sbit at 0xD2 OV;
sbit at 0xD3 RS0;
sbit at 0xD4 RS1;
sbit at 0xD5 F0;
sbit at 0xD6 AC;
sbit at 0xD7 CY;

/* WDCON */
sbit at 0xD8 RWT;
sbit at 0xD9 EWT;
sbit at 0xDA WTRF;
sbit at 0xDB WDIF;
sbit at 0xDC PFI;
sbit at 0xDD EPFI;
sbit at 0xDE POR;
sbit at 0xDF SMOD_1;

/* EIE */
sbit at 0xE8 EX2;
sbit at 0xE9 EX3;
sbit at 0xEA EX4;
sbit at 0xEB EX5;
sbit at 0xEC EWDI;

/* EIP0 */
sbit at 0xF8 LPX2;
sbit at 0xF9 LPX3;
sbit at 0xFA LPX4;
sbit at 0xFB LPX5;
sbit at 0xFC LPWDI;

#endif
```

More Information

DS89C420: QuickView -- Full (PDF) Data Sheet -- Free Samples

DS89C430: QuickView -- Full (PDF) Data Sheet

DS89C440: QuickView -- Full (PDF) Data Sheet

DS89C450: QuickView -- Full (PDF) Data Sheet