**EE 354**
**Notes on C for the 8051 family**

*Memory:*
The typical 8051 system has 256 bytes of internal data memory, 64K bytes of internal code memory, and may have an additional 64K bytes of external data and code memory. The 89C2051 has only 128 bytes of internal data memory and only 2K bytes of internal reprogrammable code memory. The AT89C51CC03 from Atmel has 256 bytes of internal data memory and 64K bytes of reprogrammable flash code memory. In a typical system the first 256 bytes of internal data memory are indirectly addressable and the first 128 bytes are directly accessible. Internal data memory also has a small area set aside for bit addressable locations. This is typically just 16 bytes (128 bits) and is located from 20h to 2Fh.

Internal data memory can be thought of as existing in three different types:
data – this refers to the first 128 bytes and is directly accessible.
idata – this refers to the first 256 bytes and is indirectly assessible.
bdata – this is bit memory and is bit addressable.

External data memory can be thought of as existing in two different types:
xdata – this refers to the whole 64K bytes of external data space. It is also typically the address space of memory mapped I/O. It is accessed indirectly through dptr.
pdata – this refers to the first 256 bytes of external data space.

Here are some examples from the C51 Manual on data declarations in C
```
char data var1;  //var1 is in data memory
char code text[] = "ENTER PARAMETER:";  //a string in code memory
unsigned long xdata array[100];  //an array inexternal data memory
float idata x,y,z;  //floats in internal data memory
unsigned int pdata dimension; //an int in pdata
unsigned char xdata vector[10][4][4]; //a 2-D array in external data
   memory
char bdata flags; //an 8-bit variable named flags in bit memory.
```

Note that if you do not specify the memory type then variables are stored according to the memory model default that is used.

Memory models in C51 may be small, compact, or large. The small model is most often used.
small – all data variables and the stack are stored in internal data memory.
compact – all data variables are stored in pdata (the first 256 bytes of external data memory. Port 2 bits are used to address this memory.
large – all data variables are stored in external data memory and may take up to 64K bytes
To declare a memory model in your code you use a pragma directive at the beginning of your code. For example
```
#pragma small
```
sets the small memory model.

Note that if you do not specify a memory model then you get the small model by default. It's not necessary to put the #pragma statement in your code if you are using µVision 2. You can do this in your project by selecting *Project → Options for Target → Target* and select the Small memory model.

*Data Types*

| **Type** | **bits** | **bytes** | **range** |
|---|---|---|---|
| `bit` | 1 | | 0 to 1 |
| `signed char` | 8 | 1 | -128 to +127 |
| `unsigned char` | 8 | 1 | 0 to 255 |
| `enum` | 16 | 2 | -32768 to +32767 |
| `signed short` | 16 | 2 | -32768 to +32767 |
| `unsigned short` | 16 | 2 | 0 to 65535 |
| `signed int` | 16 | 2 | -32768 to +32767 |
| `unsigned int` | 16 | 2 | 0 to 65535 |
| `signed long` | 32 | 4 | -2147483648 to 2147483647 |
| `unsigned long` | 32 | 4 | 0 to 4294967295 |
| `float` | 32 | 4 | ±1.175494E-38 to ±3.402823E+38 |
| `sbit` | 1 | | 0 to 1 |
| `sfr` | 8 | 1 | 0 to 255 |
| `sfr16` | 16 | 2 | 0 to 65535 |

*Accessing Memory Mapped Ports*

The 8051 family uses memory mapped I/O and the I/O ports are carved out of the external data memory address space. In assembly langauge external data memory is accessed via the dptr register. In C51 a macro has been defined in the header file absacc.h as follows

```
   #define XBYTE ((unsigned char volatile xdata*) 0)
```

To use the macro in your program you must have the include file with the directive:

```
#include <absacc.h>
```

For example, to input from a port at 0FFCh you would write

```
   unsigned char d;
   d = XBYTE[0x0ffc];
```

Or, for output to port 0ffc:

```
   d = 0x12;
   XBYTE[0x0ffx] = d;
```

*Bit Operations*

The bit operators in C are shown in the table below. The AND, OR, and exlcusive OR operators do the same thing that their assembly language counterparts do. In assembly language you can logically AND a register with the accumulator as in: `anl a, r1` In C you can logically AND two variables together as in `a = a & b;` The same can be done with the OR operator and the Exclusive OR operator.

Note that C also supports logical operators that do not operate on bit but instead, operate on Boolean values of true and false. The logical AND is a double ampersand and the logical OR is the double vertical line. These are most commonly used in IF statements as in `if(a < 7 && b >= 14 || c == 2)`

| **Op** | **Function** |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise Exclusive OR |
| << | Left shift |
| >> | Right shift |
| ~ | One's complements (inverse) |

There is no rotate operator in C. You can only do a left or a right shift. For example `a << 4`; shifts the variable a to the left four places.

The tilde symbol (~) does the inverse operation on a single variable. For example if a is a variable with a value of 0FFH then ~a would be 00H. Note that there is also a logical inverse operator which is the exclamation mark which operates on logical variables or constants. For example we might say `if(!(a >= 7))` which is the same as a < 7. It's easy to confuse && with &, || with | and ~ with ! so be careful when using these operators. In keeping with its user hostile reputation, any of these operators compile without syntax errors in place of the other but the results are not the same.

*Sample Programs*

```
//Bits
//This program illustrates the use of sbit, sfr, and bit types
unsigned char bdata x;   //sbit and bdata must be global
sbit bit0 = x ^ 0;       //bdata is bit data location 20h to 2fh
sbit bit7 = x ^ 7;       //  in internal data memory
sfr P0 = 0x80;           //special function registers for ports
sfr P1 = 0x90;           //  sfr range from 0x80 to 0xff
sfr P2 = 0xA0;
sfr P3 = 0xB0;
sbit P1_0 = P1^0;        //Can define sbits in ports
sbit P1_2 = P1^2;
sbit P1_3 = P1^3;

void main()
  {bit y;
   while(1)
    {bit0 = 1;   //bit0 and bit7 are globally defined lsb and msb
     bit7 = 1;   //  of the char variable x
     P0 = 0xff;  //output to ports
     P2 = 0xaa;
     P1 = 0xff;
     P1_0 = 0;   //set port bits
     P1_2 = 0;
     x = P1_0;   //set x to bit 0 of port 1
     y = x;      //set bit y to bit x
     P1_3 = y;   //set bit 3 of port 1 to y
    }
  }
```

---

```
//Delay.c
//This program illustrates a software delay.
#include <reg51f.h>
void Delay(unsigned char);
void main()
  {while(1)
     Delay(1);
  }
//
void Delay(unsigned char x)
  {int c;
   unsigned char i;
   for(i=0;i<x;i++)        //for loops nested x times
     for(c=0;c<1000;c++);
   return;
  }
```

```
//LongInt.c
// This program inputs 4 bytes from P0, adds them up, and finds
// the average.  The average is converted to BCD and displayed
// on P2 and P1.  Long ints are used to preserve precision
// so that the display always gives 3 digits for the average
// in the form of ddd., dd.d, or d.dd
#include<reg51.h>
void Convert2BCD(long avg);
void main(void)
  {unsigned char w, x, y, z;
   long sum, avg;
   w = P0;           //Input 4 bytes as unsigned chars
   x = P0;
   y = P0;
   z = P0;
   sum = w + x + y + z;  //Add them up in a long;
   avg = 100*sum/4;      //*100 and divide by 4 for avg
                         //  this preserves precision
   Convert2BCD(avg);     //Convert to BCD and display
  }
//
void Convert2BCD(long avg)
  {unsigned char d0, d1, d2, d3, d4;
   d0 = avg % 10;  //Convert to BCD by using mod function
   avg = avg/10;   // Largest avg = 255*100 = 25500 so
   d1 = avg % 10;  // five digits
   avg = avg/10;
   d2 = avg % 10;
   avg = avg/10;
   d3 = avg % 10;
   avg = avg/10;
   d4 = avg %10;
   if(d4 != 0)     //If d4 is not 0 display d4,d3,d2
     {P2 = d4;     //  decimal at the end
      P1 = d2 + d3*16;
       }
   else if(d3 != 0) //If d4 = 0 but d3 != 0 then display
     {P2 = d3;      //  d3 d2.d1
        P1 = d1 + d2*16;
       }
   else             //if d4 = 0 and d3 = 0 then display
     {P2 = d2;      //  d3.d2,d1
        P1 = d0 + d1*16;
       }
  }
```

```
//MemPort.c
//This program illustrates use of a port in memory.  Use Debug to see
// the port change.
#include <reg51f.h>
#include <absacc.h>
#define IO_IN 0xA000
#define IO_OUT 0xA000

void main()
  {unsigned char x, y;
   x = 1;
   while(1)
     {XBYTE[IO_OUT] = x;
      x++;
      y = XBYTE[IO_IN];
     }
  }
```

```
//Shift.c
//This program illustrates shifting and rotating.
#include <reg51f.h>
sbit b1 = 0x91;
void main()
  {unsigned char x;
   x = 3;
   b1 = 1;
   while(1)
     {x = x << 1;
        if(x == 0x80)
          x = 0x81;
        if(x == 2)
          x = 3;
        P1 = x;
      P1 = P1 & b1;
     }
  }
```