

Keil C51 Cross Compiler

- ANSI C Compiler
 - Generates fast compact code for the 8051 and its derivatives
- Advantages of C over Assembler
 - Do not need to know the microcontroller instruction set
 - Register allocation and addressing modes are handled by the compiler
 - Programming time is reduced
 - Code may be ported easily to other microcontrollers
 - (not 100% portable)
- C51 supports a number of C language extensions that have been added to support the 8051 microcontroller architecture e.g.
 - Data types
 - Memory types
 - Pointers
 - Interrupts

C vs Assembly Language

Code efficiency can be defined by 3 factors: -

1. How long does it take to write the code
 2. What size is the code? (how many Bytes of code memory required?)
 3. How fast does the code run?
- C is much quicker to write and easier to understand and debug than assembler.
 - Assembler will normally produce faster and more compact code than C
 - Dependant on C compiler
 - A good knowledge of the micro architecture and addressing modes will allow a programmer to produce very efficient C code

C51 Keywords

Keywords

To facilitate many of the features of the 8051, the C51 compiler adds a number of new keywords to the scope of the C language:

**_at_
alien
bdata
bit
code
compact
data**

**far
idata
interrupt
large
pdata
priority
reentrant**

**sbit
sfr
sfr16
small
task
using
xdata**

C51 Data Types

Data Types	Bits	Bytes	Value Range
bit †	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	8 / 16	1 or 2	-128 to +127 or -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	$\pm 1.175494\text{E-}38$ to $\pm 3.402823\text{E}+38$
sbit †	1		0 or 1
sfr †	8	1	0 to 255
sfr16 †	16	2	0 to 65535

† = extension to ANSI C

C51 Memory Models

- Small
 - All variables stored in internal data memory
 - Be careful – stack is placed here too
 - Generates the fastest, most efficient code
 - Default model for Keil uVision Projects
- Compact
 - All variables stored in 1 page (256 bytes) of external data memory
 - Accessed using MOVX @R0
 - Slower than small model, faster than large model
- Large
 - All variables stored in external data memory (up to 64KByte)
 - Accessed using MOVX @DPTR
 - Generates more code than small or compact models

C51 Memory Models

Select Model

Options for Target 'Target 1'

Device Target Output Listing C51 A51 BL51 Locate BL51 Misc Debug Utilities

Intel 8051AH

Crystal (MHz): 12.0 Use On-chip ROM (0x0-0xFFFF)

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: None

Off-chip Code memory

	Start:	Size:
Eprom		
Eprom		
Eprom		

Off-chip Xdata memory

	Start:	Size:
Ram		
Ram		
Ram		

Code Banking

Banks: 2 Bank Area: 0x0000 0xFFFF

Start: End:

'far' memory type support

Save address extension SFRs in interrupts

OK Cancel Defaults

C51 Memory Types

- Memory type extensions allow access to all 8051 memory types.
 - A variable may be assigned to a specific memory space
 - The memory type may be explicitly declared in a variable declaration
 - *variable_type <memory_type> variable_name;*
 - *e.g. int data x;*
- Program Memory
 - CODE memory type
 - Up to 64Kbytes (some or all may be located on 8051 chip)
- Data Memory
 - 8051 derivatives have up to 256 bytes of internal data memory
 - Lower 128 bytes can be directly or indirectly addressed
 - Upper 128 bytes shares the same address space as the SFR registers and can only be indirectly addressed
 - Can be expanded to 64KBytes off-chip

C51 Memory Types

- **code**

- Program memory (internal or external).
- *unsigned char code const1 = 0x55; //define a constant*
- *char code string1[] = "hello"; //define a string*

- **data**

- Lower 128 bytes of internal data memory
- Accessed by direct addressing (fastest variable access)
- *unsigned int data x; //16-bit variable x*

- **idata**

- All 256 bytes of internal data memory (8052 micro)
- Accessed by indirect addressing (slower)

C51 Memory Types

- **bdata**
 - Bit addressable area of internal data memory (addresses 20H to 2FH)
 - Allows data types that can be accessed at the bit level
 - *unsigned char bdata status;*
 - *sbit flag1 = status^0;*
- **xdata**
 - External data memory
 - Slower access than internal data memory
 - *unsigned char xdata var1;*

8051 Memory Usage

- In a single chip 8051 application data memory is scarce
 - 128 or 256 bytes on most 8051 derivatives
- Always declare variables as the smallest possible data type
 - Declare flags to be of type bit
 - Use chars instead of ints when a variable's magnitude can be stored as 8 bits
- Use code memory to store constants and strings

Example Code (1)

```
sbit input = P1^0;  
void main()  
{  
    unsigned char status;  
    int x;  
  
    for(x=0;x<10;x++)  
    {  
        status = input;  
    }  
}
```

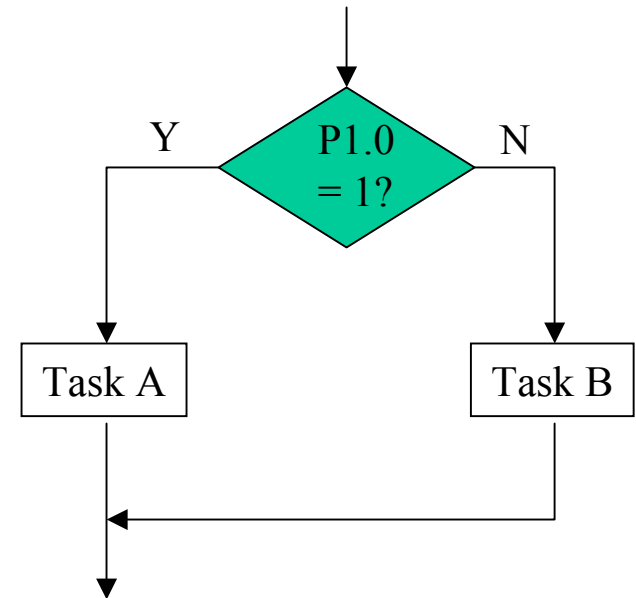
- How can the variable declarations be improved?

Program Branching

- Most embedded programs need to make some decisions
 - e.g. perform a certain task only if a switch input is high
- An if-else statement is the most common method of coding a decision box of a flowchart.

```
if (P1^0 == 1)
{
    //task A
}
else
{
    //task B
}
```

- Note that the brackets are only required if the if or else statement contains more than 1 line of code



Wait Loops

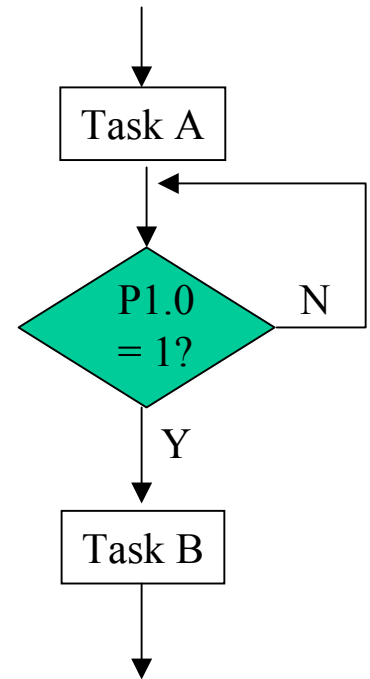
- In some applications the program must wait until a certain condition is true before progressing to the next program task (wait loop)
 - e.g. in a washing machine wait for the water to heat before starting the wash cycle
- A while statement is often used here

//task A

while(P1^0 == 0); //will stay here while P1.0 is low

//can also write as while(P1^0 == 0) {}

//task B



Continuous Loops

- An embedded program never ends
 - It must contain a main loop than loops continuously
 - e.g. a washing machine program continually tests the switch inputs
- A continuous loop is programmed using a loop with no condition
 - *while(1){ }*
 - *while(1);*
 - *for(;;){ }*
 - *for(;;);*

C Bitwise Operators

- NOT \sim
char data x = 0x05;
x = ~x; //x=0xFA
- AND $\&$
char data x = 0x45;
x &= 0x0F; //x = 0x05
 - Useful for clearing specific bits within a variable (masking)
- OR $|$
char data x = 0x40;
x |= 0x01; //x = 0x41
 - Useful for setting individual bits within a variable
- EXCLUSIVE OR \wedge
char data x = 0x45;
x ^= 0x0F; //x = 0x4A
 - Useful for inverting individual bits within a variable

Question

- What is the value of variable x after the following code executes?

unsigned char data x;

x = 21;

x &= 0xF0;

x |= 0x03;

- Write C code to set the upper 4 bits of a char to “1100”.

C Logical Operators

- Bitwise operators will change certain bits within a variable
- Logical operators produce a true or false answer
 - They are used for looping and branching conditions
- C Logical Operators
 - AND &&
 - OR ||
 - Equals ==
 - Not Equal !=
 - Less <
 - Less or Equal <=
 - Greater >
 - Greater or Equal >=

Example Code (2)

```
//program to add 2 8-bit variables  
//a flag should be set if the result exceeds 100  
void main()  
{  
    unsigned char data num1, num2;  
    unsigned int data result;  
    bit overflow;  
  
    num1 = 10;  
    num2 = 25;  
    result = num1 + num2;  
    if (result > 100)  
        overflow = 1;  
}
```

Accessing Port Pins

- Writing to an entire port

```
P2 = 0x12;           //Port 2 = 12H (00010010 binary)
```

```
P2 &= 0xF0;        //clear lower 4 bits of Port 2
```

```
P2 |= 0x03;        //set P2.0 and P2.1
```

- Reading from a port

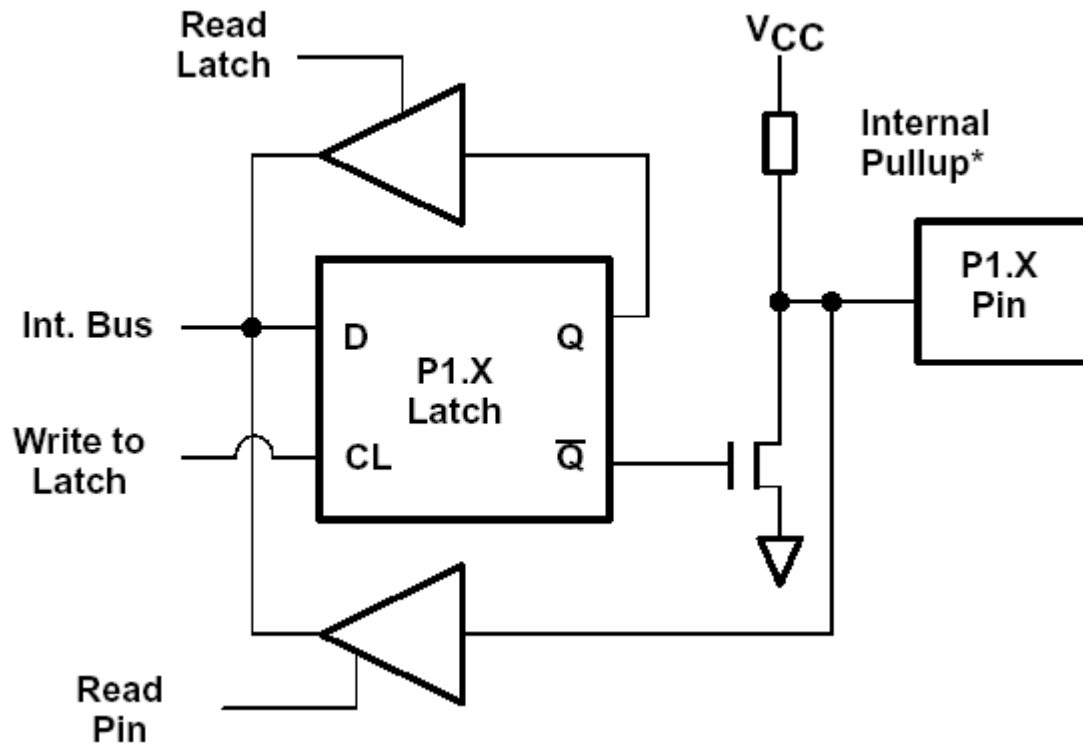
```
unsigned char data status;
```

```
status = P1;       //read from Port 1
```

- Before reading from a port pin, always write a ‘1’ to the pin first.

Reading 8051 Port Pins

- We can only read the proper logic value of a pin if the latch contains a '1'.
 - If the latch contains a '0' the FET is on and a '0' will always be read.



Example Code (3)

*//Program to read from 8 switches connected to Port 1. The status of the switches
//is written to 8 LEDs connected to Port 2.*

```
#include <reg51.h>                //SFR address definitions
void main()
{
    unsigned char data status;      //variable to hold switch status

    //continuous loop to read switches and display the status
    while(1)
    {
        status = P1;
        P2 = status;
    }
}
```

Accessing Individual Port Pins

- Writing to a port pin

```
//write a logic 1 to port 0 pin 1
```

```
P0^1 = 1;
```

- In a program using lots of individual port pins it is better coding practice to name the pins according to their function

```
sbit power_led = P0^1;
```

```
power_led = 1;
```

- Reading from a port pin

```
//Turn on LED if switch is active
```

```
sbit switch_input = P2^0;
```

```
if (switch_input)
```

```
    power_led = 1;
```

```
else
```

```
    power_led = 0;
```

Example Code (4)

//program to flash an LED connected to Port 2 pin 0 every 0.2 seconds

```
#include <reg51.h>
```

```
sbit LED = P2^0;
```

```
void delay();
```

```
void main()
```

```
{
```

```
    while (1)
```

```
    {
```

```
        LED = 0;    //LED off
```

```
        delay();
```

```
        LED = 1;    //LED on
```

```
        delay();
```

```
    }
```

```
}
```

```
//Delay function
```

```
void delay()
```

```
{
```

```
.....
```

```
}
```

Generating Delays

- Software Delays
 - Uses looping mechanisms in C or assembly
 - Does not use any microcontroller hardware resources
 - Ties up the CPU while the delay is running
 - Delay produced depends on the compiler
- Hardware Delays
 - Uses a microcontroller timer
 - Uses very little CPU resources (runs in background)

Software Delay

- Use a for loop
 - Does not use any timer resources
 - Uses CPU resources while running
 - i.e. no other task can be performed during delay loop
 - Can result in large amounts of machine code being generated
 - Results may be different for different compilers
- The following code results in a 204 usecond delay for the 8051 operating off the 12MHz oscillator
- For loops can be nested to produce longer delays

```
void delay()  
{  
    unsigned char x;  
    for (x=100; x > 0; x--);  
}
```

Software Delay

```
line level    source
 1           #include <REG51.H>
 2           sbit OUTPUT = P1^0;
 3           void main()
 4           {
 5     1       unsigned char x;
 6     1       while(1)
 7     1       {
 8     2           OUTPUT = ~OUTPUT;
 9     2           for(x=100;x>0;x--);
10    2       }
11    1       }
12
!C51 COMPILER V7.10  DELAY_FOR_LOOP
ASSEMBLY LISTING OF GENERATED OBJECT CODE

                ; FUNCTION main (BEGIN)
                ; SOURCE LINE # 3
                ; SOURCE LINE # 4
0000            ?C0001:
                ; SOURCE LINE # 6
                ; SOURCE LINE # 7
                ; SOURCE LINE # 8
0000 B290            CPL        OUTPUT
                ; SOURCE LINE # 9
;----- Variable 'x' assigned to Register 'R7' -----
0002 7F64            MOV        R7,#064H
0004            ?C0003:
0004 DFFE            DJNZ      R7,?C0003
0006 80F8            SJMP      ?C0001
                ; FUNCTION main (END)
```

Using a decrementing for loop will generate a DJNZ loop.

Loop execution time =
204 machine cycles
[1 + 1 + (100 * 2) + 2]
= 204usec for a 12MHz crystal.
Change starting value to 98 to
get a 200usec delay

Hardware Delay

- Use timer 0 or timer 1
 - Very efficient mechanism of producing accurate delays
 - Timer mode, control and counter registers must be configured
 - Timer run bit is used to start/stop the timer
 - The timer flag can be polled to determine when required time delay has elapsed
 - Using timer in interrupt mode uses very little CPU resources
- See timer notes for more details

C51 Functions

- Can specify
 - Register bank used
 - Memory Model
 - Function as an interrupt
 - Re-entrancy
- *[return_type] func_name([args]) [model] [re-entrant] [interrupt n] [using n]*

```
int square(int x)  
{  
    return(x * x);  
}
```

Re-entrant Functions

- A re-entrant function is a function that can be called while it is still running.
 - e.g. an interrupt occurs while the function is running and the service routine calls the same function.
- Keil compiler supports re-entrant functions.
 - Beware of stack limitations

Scope of Variables

- A variable defined within a function will default to an **automatic** variable
 - An automatic variable may be overlaid i.e. the linker may use the variable's memory space for a variable in another function call.
 - This will cause the variable to lose its value between function calls.
- If you want a variable to maintain its value between function calls, declare it as a **static** variable.
 - *static int x;*
 - Static variables cannot be overlaid by the linker
- Declare a variable as **volatile** if you want its value to be read each time it is used

Function Parameter Passing

- Up to 3 arguments may be passed in registers
 - This improves system performance as no memory accesses are required
 - If no registers are available fixed memory locations are used

Argument Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R6 & R7	R4—R7	R1—R3
2	R5	R4 & R5	R4—R7	R1—R3
3	R3	R2 & R3		R1—R3

Function Return Values

- Function return values are always passed in registers

Return Type	Register	Description
bit	Carry Flag	
char, unsigned char, 1-byte ptr	R7	
int, unsigned int, 2-byte ptr	R6 & R7	MSB in R6, LSB in R7
long, unsigned long	R4-R7	MSB in R4, LSB in R7
float	R4-R7	32-Bit IEEE format
generic ptr	R1-R3	Memory type in R3, MSB R2, LSB R1

Interrupt Functions

- Interrupt vector number is provided in the function declaration
 - *void timer0_int() interrupt 1 using 2*
{
....
}
- *Contents of A, B, DPTR and PSW are saved on stack when required*
- *All working registers used in the ISR are stored on the stack if the using attribute is not used to specify a register bank*
- *All registers are restored before exiting the ISR*

Interrupt Number	Interrupt Description	Address
0	EXTERNAL INT 0	0003h
1	TIMER/COUNTER 0	000Bh
2	EXTERNAL INT 1	0013h
3	TIMER/COUNTER 1	001Bh
4	SERIAL PORT	0023h

Absolute Variable Location

- The `_at_` keyword is used to assign an absolute memory address to a variable
 - Useful for accessing memory mapped peripherals or specific memory locations
 - Syntax
 - `type [memory_space] variable_name _at_ constant`
 - *`unsigned char xdata lcd _at_ 0x8000`*
 - Absolute variables may not be initialised

C51 Pointers

- Generic pointers
 - Same as ANSI C pointers
 - *char *ptr;*
 - Stored using 3 bytes
 - Can be used to access any memory space
 - Code generated executes more slowly than memory specific pointers
- Memory Specific Pointers
 - Specify the memory area pointed to
 - *char data *ptr1; //pointer to char in data memory*
 - Stored as 1 (data, idata) or 2 bytes (code or xdata)
 - Can also specify where pointer is stored
 - *char data *xdata ptr2; //pointer to data stored in xdata*