Embedded C programming using Keil

What is Embedded C?

The C programming language was originally designed for computer and not embedded systems. For programming microcontrollers used in embedded systems, support for direct access to different registers and reading and setting of bits are important. This support was not provided by C language.

Embedded C is a set of language extensions for the C programming language by the C standards Committee to cater to the programming of Embedded systems.

Difference between C and Embedded C?

- C is used for desktop computers while embedded C is used for microcontroller based applications.
- C has the luxury to use resources of a desktop computer like memory, OS etc. Embedded C has to use limited resources (RAM, ROM, I/O's) on an embedded processor.
- Compiler for C typically generates OS dependant executables. Embedded C requires compilers to create files to be downloaded to the microcontroller/microprocessor where it needs to run.

Introduction to Keil compiler

Keil Software (http://www.keil.com/) publishes one of the most complete development tool suites for 8051 software, which is used throughout industry. For development of Embedded C code, their Developer's Kit product includes their C51 compiler, as well as an integrated 8051 simulator for debugging.

Limitations of the Keil compiler and the modifications it has made to the C language (Embedded C) to develop software for the 8051 architecture based microcontrollers is explained in the following sections.

Keil Limitations

There are several very important limitations in the evaluation version of Keil's Developer's Kit that users need be aware of when writing software for the 8051.

- **Object code must be less than 2 Kbytes**

  The compiler will compile any-sized source code file, but the final object code may not exceed 2 Kbytes. If it does, the linker will refuse to create a final binary executable (or HEX file) from it. Along the same lines, the debugger will refuse any files that are over 2Kbytes, even if they were compiled using a different software package.

- **Program code starts at address 0x800**

  All C code compiled and linked using the Keil tools will begin at address 0x800 in code memory. Such code may not be programmed into devices with less than 2Kbytes of Read-Only Memory. Code written in assembly may circumvent this limitation by using "origin" keyword to set the start to address 0x0000. No such work-around exists for C programs.

C Modifications

The Keil C compiler has made some modifications to the ANSI-compliant implementation of the C programming language. These modifications were made solely to facilitate the use of a higher-level language like C for writing programs on microcontrollers. Some of the modifications are mentioned below.

- **Variable types**

  The Keil C compiler supports most C variable types and adds several of its own.

  - **Standard types**

    The evaluation version of the Keil C compiler supports the standard ANSI C variable types, with the exception of the floating point types. These types are summarized below.

Type	Bits	Bytes	Range
char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32,768 to +32,767
short	16	2	-32,768 to +32,767
unsigned short	16	2	0 to 65,535
int	16	2	-32,768 to +32,767
unsigned int	16	2	0 to 65,535
long	32	4	-2,147,483,648 to +2,147,483,647
unsigned long	32	4	0 to 4,294,697,295

    In addition to these variable types, the compiler also supports the **struct** and **union** data structures, as well as type redefinition using **typedef**.

Some of the standard data types are explained below

1. **Unsigned char (** Range:0-255 **)**
   Since 8051 is an 8-bit microcontroller, the character data type is the most natural choice for many applications. The compiler uses signed char as default if we do not put the keyword unsigned in front of the char. Example:

```
void main(void){
 unsigned char z;
 for(z=0,z<=255;z++){
       P1 =z}
  }
```

2. **Signed char (**Range:- -128 to-127**)**
   The signed char is an 8-bit data type that uses the most significant bit to represent the sign of a number (+ or -). As a result we have only 7 bit for the magnitude of the signed number.

3. **Unsigned int (**Range:- 0 to 65535)
   The unsigned int is a 16-bit data type that takes a value in the range of 0 to 65535. In 8051, unsigned int is used to define 16 bit variables such as memory addresses.
   It should be noted again that the C compiler uses signed int as the default if we do not use the keyword unsigned.

4. **Signed int (**Range: -32768 to +32767**)**
   Signed int is a 16 bit data type that uses the most significant bit to represent the sign of the number (+ or -). As a result, we have only 15 bits for the magnitude of the number, or values from -32768 to 32767.

- **Keil Types**

To support a microcontroller and embedded systems applications, Keil added several new types to their compiler. These are summarized in the table below.

Type	Bits	Bytes	Range
bit	1	0	0 to 1
sbit	1	0	0 to 1
sfr	8	1	0 to 255

1. **bit**

This is a data type that gets allocated out of the 8051's bit-addressable on-chip RAM. Like other data types, it may be declared as a variable. However, unlike standard C types, it may not be used as a pointer. An example of its usage follows.

```
/* declare two bit variables - the compiler will
decide which */
/* addresses they are at.  Initialize them to 0 and
1. */
bit testbit1 = 0;
bit testbit2 = 1;

/* set testbit1 to the value in testbit2 */
testbit1 = testbit2;

/* clear testbit2 */
testbit2 = 0;

/* testbit1 is now a 1, and testbit2 is now a 0 */
/* Note that the assignment of testbit2 to testbit1
only copied */
/* the contents of testbit2 into testbit1.  It did
*not* change */
/* the location of testbit1 to be the same as
testbit2. */
```

2. **sbit (Single bit)**

sbit keyword is a widely used 8051 C data type designed specifically to access single bit addressable registers. It allows access to the single bit of the SFR registers. The SFR's for the ports P0-P3, which are widely used, are bit addressable. An example of the usage of sbit is shown below.

```
#include<AT89C5131.h>

sbit LED = P2^0;//Declaration should be outside main()

void main(void){

while(1){

    LED = 0;

    delay();
```

```
    LED = 1;

    delay();

}

}

void delay(void){

                    unsigned int i;

                    for( i=0;i<50000;i++);

                  }

      /*Port2 pin 0 is addressed using the sbit
declaration.

       This port pin is toggled after a specific delay
*/
```

3. **sfr**

sfr type defines a special function registers. For example

```
sfr P0 =  0x80;
sfr P1 =  0x90;
sfr P2 =  0xA0;
sfr P3 =  0xB0;
```

P0, P1, P2 and P3 are the SFR name declarations. Names for the sfr variables are defined just like other C variable declarations. Any symbolic name can be used in an sfr declaration. Classic 8051 devices support the SFR address range 0x80 – 0xFF. sfr variables may not be declared inside a function. The must be declared outside of the function body.

- **Keil variable Extensions**

In writing applications for a typical computer, the operating system manages memory on behalf of the programs, eliminating their need to know about the memory structure of the hardware. Even more important, most computers having a unified memory space, with the code and data sharing the same RAM. This is not true with the 8051, which has separate memory spaces for code, on-chip data, and external data.

To accommodate for this when writing C code, Keil added extensions to variable declarations to specify which memory space the variable is allocated from, or points to. The most important of these for programmers are summarized in the following table.

Extension	Memory Type	Related ASM
data	Directly-addressable data memory (data memory addresses 0x00-0x7F)	MOV A, 07Fh
idata	Indirectly-addressable data memory (data memory addresses 0x00-0xFF)	MOV R0, #080h MOV A, @R0
xdata	External data memory	MOVX @DPTR
code	Program memory	MOVC @A+DPTR

Example: Accessing the code data space in 8051 C using the code extension

Example:

```
#include<AT89C5131.h>

void main(void)

{

code unsigned char mydata[]= "Hello";

unsigned char z;

for(z=0;z<=5;z++){

                    P1 = mydata[z];

                  }

}
```

This program uses a separate area of the code space for data. This allows the size the array to be as long as you want if you have enough on chip ROM. However, more the code space you use for data, the less is the space left for the program code.

- **Keil Function Extensions**

  As in most other C compilers, functions may be declared as shown below

```
unsigned int <function name> (unsigned int var)
{
    ....
    return (var);
}
```

Keil provides important extensions to the standard function declaration to allow for the creation of interrupt handlers and reentrant functions.

- **interrupt**

In writing applications for a typical computer, the operating system provides system calls for setting a function, declared in the standard manner, as the handler for an interrupt. However, in writing code for an 8051 without an operating system, such a system would not be possible using solely C code. To eliminate this problem, the Keil compiler implements a function extension that explicitly declares a function as an interrupt handler. The extension is **interrupt**, and it must be followed by an integer specifying which interrupt the handler is for. For example:

```
/* This is a function that will be called whenever
a serial interrupt occurs. Note that before this
will work, interrupts must be enabled.*/
void serial_int (void) interrupt 4
{
    ...
            }
```

In the example above, a function called **serial_int** is set as the handler for interrupt 4, which is the serial port interrupt. The number is calculated by subtracting 3 from the interrupt vector address and dividing by 8. The five standard interrupts for the 8051 are as follows:

Interrupt	Vector address	Interrupt number
External 0	0003h	0
Timer 0	000Bh	1
External 1	0013h	2
Timer 1	001Bh	3
Serial	0023h	4

Other interrupts are dependent on the implementation in the particular 8051-derivative being used in the project, but may be calculated in the same manner using the vector addresses specified by the manufacturer.

- **Using**

Since the processor only save the current program counter before executing an interrupt handler, the handler can potentially damage any data that was in the registers prior to the interrupt. This in turn would corrupt the program once the processor goes back to where it left off. To avoid this, the Keil compiler

determines which registers will be used by the interrupt handler function, pushes them out to the stack, executes the handler, and then restores the registers from the stack, before returning to the interrupted code. However, this incurs extra time, especially if a lot of registers will be used. It is preferred that as little time be spent in interrupts as possible. To decrease this time, Keil provides an optional extension, **using**, to the **interrupt** extension that tells the compiler to change to a new register bank prior to executing the handler, instead of pushing the registers to the stack.

```
/* This is a function that will be called whenever a
serial */
/* interrupt occurs. Prior to executing the handler, the
*/
/* processor will switch to register bank 1
void serial_int (void) interrupt 4 using 1
{
    ...
```

In the 8051, interrupts have two possible priorities: high and low. If, during the processing of an interrupt, another interrupt of the same priority occurs, the processor will continue processing the first interrupt. The second interrupt will only be processed after the first has finished. However, if an interrupt of a higher priority arrives, the first (low priority) interrupt will itself be interrupted, and not resume until the higher priority interrupt has finished. Because of this, all interrupts of the same priority may use the same register bank

- **Reentrant**

Similar to the case described for interrupts above, it is possible for a single function to be interrupted by itself. For example, in the middle of normal execution of the function, the interrupt occurs, and that interrupt makes a call to the same function. While the interrupt handler will save the registers before entering this function, no protective measures are taken from overwriting the contents of local variables allocated in data memory. When the interrupt is serviced and control is passed back to normal execution, the corrupted data in those variables could ruin the entire program.

The general term for a function that may be called more than once simultaneously is "reentrant." Accordingly, the **reentrant** extension may be used in a function declaration to force the compiler to maintain a separate data area in memory for each instance of the function. While safe, this does have the potential to use large area of the rather limited data memory. An example of such a function follows.

```
/* Because this function may be called from both the main
program */
/* and an interrupt handler, it is declared as reentrant
to */
/* protect its local variables. */

int somefunction (int param) reentrant
{
    ...
    return (param);
}

/* The handler for External interrupt 0, which uses
somefunction() */
void external0_int (void) interrupt 0
{
    ...
    somefunction(0);
}

/* the main program function, which also calls
somefunction() */
void main (void)
{
    while (1==1)
    {
        ...
        somefunction();
    }
}
```

- **Operators in C**

The compiler supports the following operators,

- Logical operators: AND (&&), OR (||) and NOT (!)
- Bitwise operators: AND(&), OR(|),XOR(^), Inverter(~)
- Shift operators: Shift Right(>>), Shift Left(<<)

Bitwise operators for C

		AND	OR	XOR	INVERTER
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Examples using C bitwise operators

1. 0x35 & 0x0F = 0x05 /*ANDing*/
2. 0x04 | 0x68 = 0x6C /*ORing*/
3. 0x54 ^ 0x78 = 0x2C /*XORing*/
4. ~0x55 = 0xAA /*Inverting*/

The shift right(>>) and the shift left(<<) operators have the following format in C.

data >> number of bits to be shifted right
data << number of bits to be shifted left

1. 0x9A >>3 = 0x13 /* Shift right 3 times */
2. 0x77 >> 4 = 0x07 /* Shift right 4 times */
3. 0x06 << 4 = 0x60 /*Shift left 4 times */

- **Sample C programs**

  - **I/O programming examples**

```
#include<AT89C5131.h>

void main(void){

unsigned char mybyte;

P0 = 0xFF;

while(1){

    mybyte = P0;

    if(mybyte < 100)

          P1 = mybyte;

    Else

          P2 = mybyte

}

}

/* Port 0 is configured as an input port. This port is
read continuously and assigned to P1 if the value is less
than 100 else the value is output on P2 */
```

```
#include<AT89C5131.h>

sbit inbit = P1^0

sbit outbit = P2^7

bit membit =27H

void main(void){

while(1){

membit= inbit;

outbit = membit;

}

}
```

```
/* sbit data type is used for input */
#include<AT89C5131.h>

sbit switch = P1^7

void main(void){

while(1){

 if(switch == 1){

       P0 = 0x55;

else

       P2 = 0XAA;

}

}
/* Switch variable corresponds to PORT 1 Pin7. It is
used as an input. We put 0x55 on P0 if the P1.7 status
is 1 else we put 0xAA on P2*/
```

- **Time Delay**

  There are two ways to generate time delay in 8051 C

  1. Using a simple for loop
  2. Using the 8051 timer.

  Sample program to generate delay using for loop is given below

```
#include<AT89C5131.h>

void msdelay(unsigned int){

void main(void){

while(1){

P1= 0x55;

msdelay(250)

P1 = 0XAA;

msdelay(250);

}

}
```

/* P1 keeps toggling between 0x55 and 0xAA after a delay generated by the msdelay() function */

```
void msdelay(unsigned int time){

unsigned int i,j;
for(i=0;i<time;i++)
   for(j=0;j<1275;i++);

}
```

/* Generates delay using two for loops */