In Line and Embedded Assembly
The ARM 7 C-compiler from Keil allows you to do inline assembly using the __asm directive. The __asm directive allows you to imbed assembly code inside your C program for the ARM processor. The directive has two forms:

```
__asm{mov r1, r2; first form}
```

or

```
__asm
{mov r1, r2;
 mov r3, r2;
}
```

Note that the Keil compiler is picky about where the braces go. You would think you could write

```
__asm{mov r1, r2;
 mov r3, r2;}
```

but this generates a syntax error because the semicolon in assembler indicates the end of a line. Inline assembler is not very useful the the ARM processor since it does not allow direct access to registers. All of the registers become "virtual registers" so that if you use r1 it may or may not be the actual register r1.

Embedded assembly is more useful. Embedded assembly places an assembly function inside a C program. The example below shows how to use an assembly function along with C code to output a ramp to the D to A converter.

```
#include<LPC213x.H>
__asm void DtoA(int i)
{//i is in r0
  ldr r1, =__cpp(0xE006C000);  DACR register address to r1
  lsl r2, r0, #6;  Shift r0 (i) left 6 places and put in r2
  str r2, [r1];  Output r2 to D to A
  bx lr;  Return
}
int main()
{int i;
 IODIR1 = 0xFFFF000;       // P0.25 set to DA Out
  PINSEL1 = 0x00080000;    // P0.25 set to DA Out
  while(1)
  {for(i=0;i<1024;i++)
    DtoA(i);
  }
}
```

Example of embedded assembly language
Note that parameters are passed in registers but the registers used depend on what registers the assembly program is using at the time of the call. So it becomes easier to create a global variable (in this case int i). The assembly program can access the address of i and move its data into register r2. The __cpp directive allows access to constants or constant expressions (but not variables) in the c code. The variable i is not accessible in the assembly code the its address is.

Embedded assembly is occasionally useful for sections of code that require lots of bit manipulation or that may require the fastest possible speed. For example the ARM processor has a barrel shifter that is available in hardware which is not used by the C compiler. It could be accessed by way of embedded assembly code.

Floating Point (Math) Library
To do programs in floating point arithmetic you need only declare variables of type float in your C code. If you want to do floating point functions then you need to include the math.h library in your code. For example:

```c
#include<LPC213X.H>
#include<math.h>
const float PI = 3.14159265;    //A global constant
void main()

(float x, f, t;
  t = 2.4;
  f = 60;
  x = sin(2*PI*f*t);
  //At this point you could scale x, convert it
  //  to an int and output it to the D to A
}
```

Floating point arithmetic example.

Scaling the D to A Converter
The D to A converter is a 10-bit converter which converts a 10-bit digital number to an analog voltage between a minimum (usually 0) and a maximum (usually 3.3 volts). The converter is not signed so that the digital number 00 0000 0000 produces the minimum voltage and 11 1111 1111 produces the maximum. Effectively, the D to A converter accepts binary numbers in the range 0 to 1023 so any signal that you produce digitally must be scaled to this range before you send it out.

For example, if you wanted to output a sinusoid which varied between -1 and +1 you might want -1 to be the binary number 00 0000 0000 and +1 to be the binary number 11 1111 1111. The amplitude range of the sinusoid is +2 and this needs to span the range 0 to 1023 so you must multiply the sinusoid by 1024/2. However, before you do this you must add an offset to get rid of the negative numbers. For example, if x is a number that ranges from -1 ≤ x < +1 then 512*(x + 1) would range from 0 to 1023.

Class Exercise 4-1: Develop an equation that can be used to output a sinusoid with an amplitude range of -1 to +1 to the 10-bit D to A converter.
**Class Exercise 4-2:** Using the equation from problem 4-1, write a C program to output a continuous sinusoid to the D to A port that has 100 samples for each sine cycle. The samples should be put out as fast as possible in a forever loop.

**Class Exercise 4-3:** Connect 8 LEDs to P1.16 through P1.23. Write a C program to blink these lights one at a time forever. Use a software delay loop to get enough time between blinks so that it is visible.

Note that each I/O port pin is 5 volt tolerant and will sink about 4 ma. You can use the following circuit to connect the LED to the I/O pins.

![Circuit Diagram for class exercise 4-3](image)

**Class Exercise 4-4:** Connect a push button switch to pin P1.24 and use the 8 LEDs from problem 4-3 for this problem. Write a C program to blink the lights one at a time in one direction if the switch is not pushed and in the other direction if the switch is pushed. You can connect the switch using the following circuit.

![Circuit Diagram for class exercise 4-4](image)

**Class Exercise 4-5:** You can output an arbitrary waveform to the D to A converter by storing sample values in a constant array (goes in code memory) and outputting the samples at a regular interval. For example the C code below titled WavArray.c outputs half a sinusoid to the D to A port. Modify this program so that it outputs an exponential saw tooth waveform similar to that shown below.
This program outputs an arbitrary waveform where the waveform samples are stored in an array.

```c
#include <LPC213X.H>

const int arr[] = {0, 316, 601, 828, 973, 1023, 973, 828, 601, 316,
                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  0};

int main(void)
{
    int i, delay;
    PINSEL1 = 0x00080000;  // P0.25 set to DA Out
    while(1)
    {
        for(i=0; i<20; i++)
        {
            DACR = arr[i] << 6;
            for(delay=0; delay<10000; delay++);  
        }
    }
}
```