Interfacing C and TMS320C6713 Assembly Language (Part-I)

Abdullah A. Wardak

***Abstract*—**This paper describes an interfacing of C and the TMS320C6713 assembly language which is crucially important for many real-time applications. Similarly, interfacing of C with the assembly language of a conventional microprocessor such as MC68000 is presented for comparison. However, it should be noted that the way the C compiler passes arguments among various functions in the TMS320C6713-based environment is totally different from the way the C compiler passes arguments in a conventional microprocessor such as MC68000. Therefore, it is very important for a user of the TMS320C6713-based system to properly understand and follow the register conventions when interfacing C with the TMS320C6713 assembly language subroutine. It should be also noted that in some cases (examples 6-9) the endian-mode of the board needs to be taken into consideration. In this paper, one method is presented in great detail. Other methods will be presented in the future.

***Keywords*—**Assembly language, high level language, interfacing, stack, argument**s.**

I. Introduction

In many real-time applications, execution-time is very important. In order to achieve that, the relevant code should be initially developed using a high-level language and then converted into assembly language, which can then be called from within the high-level language program [1], [2], [4].

The way in which compilers pass arguments among various functions in a particular micro-based system varies from one system to another [1]-[4]. Therefore, thorough understanding of how compilers pass arguments among various functions in a particular system plays an important role in interfacing high-level and assembly language. In many micro-based systems, the most efficient way of passing arguments among various functions is through stack [1]-[2]. However, the way the C compiler passes arguments from the calling function to the called function in the TMS320C6713-based environment is totally different from the way the C compiler passes arguments in a conventional microprocessor such as MC68000 [1]-[3]. Hence, it is very important for a user of the TMS320C6713-based system to properly understand and follow the register conventions and take into account the endianness of the board (see examples 6-9) when interfacing C with the C6713 assembly language subroutine.

Abdullah Wardak is currently a senior lecturer in Southampton Solent University, Faculty of Technology, School of Computing and Communications, East Park Terrace, Southampton SO14 OYN, UK (phone: 0044(0)2380319213, fax. 0044(0)2380334441, e-mail: Abdullah.wardak@solent.ac.uk).

II. Interfacing C and MC68000 Assembly

Stack of the MC68000 microprocessor plays a major role in interfacing C with the MC68000 assembly language. The MC68000 stack is used as a tool for passing various arguments from the main function in C to the MC68000 assembly language subroutines and from one assembly language subroutine to another.

When an argument is pushed onto the MC68000 stack, the stack pointer (A7) is pre-decremented by the size of the argument and then the argument is pushed onto the stack. When an argument is popped off the stack, the stack pointer (A7) is post-incremented by the size of the argument. For example, in Fig. 1 the stack pointer (A7) is pre-decremented by 4 each time an argument is pushed onto the stack. This is because each argument occupies 4 bytes on the stack. Similarly, wherever the stack pointer (A7) is pointing to, the item is popped off the stack and the stack pointer is then incremented by 4 afterwards.

Examples 1 and 2 describe the role of the MC68000 stack in interfacing the two programming languages. In example 1, the C function (**asmfunc**) is converted into MC68000 assembly language subroutine as shown in Fig. 1. The MC68000 assembly language subroutine is then called from the main function in C and the compiler pushes the arguments onto the stack in a manner presented in Fig. 1. Similarly in example 2, the equations which are used in 3-D image transformation and animation are implemented in MC68000 assembly language subroutine, **rotx** (see Fig. 2b). The implemented assembly language subroutine (**rotx**) is then called from the main function in C and the compiler pushes the arguments onto the stack in the manner shown in Fig. 2a.

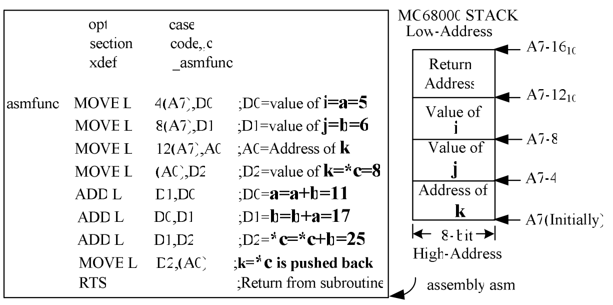
Example 1

```
#include <stdio.h>
extern  asmfunc ( );
main( )
{
    int i,j,k;
    i=5;
    j=6;
    k=8;
}   asmfunc(i, j, &k);
```

cprog.c

```
asmfunc (int a, int b, int *c)
{
    a = a + b;
    b = b + a;
    *c = *c + b;
}
```

Following is the MC68000 assembly language translation of the above C function, **asmfunc( )**.



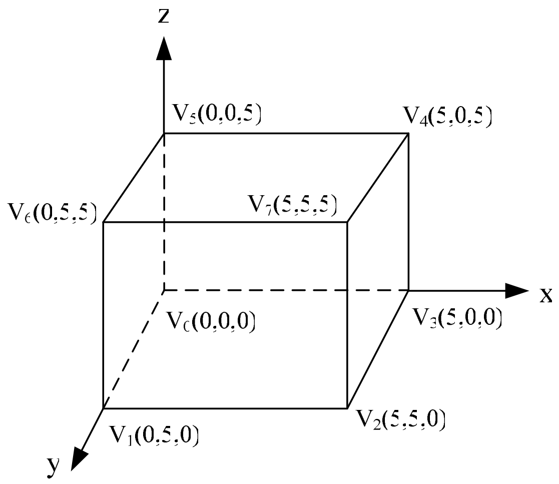
```
         opt         case
         section     code,,c
         xdef        _asmfunc

asmfunc  MOVE L      4(A7),D0     ;D0=value of i=a=5
         MOVE L      8(A7),D1     ;D1=value of j=b=6
         MOVE L      12(A7),A0    ;A0=Address of k
         MOVE L      (A0),D2      ;D2=value of k=*c=8
         ADD L       D1,D0        ;D0=a=a+b=11
         ADD L       D0,D1        ;D1=b=b+a=17
         ADD L       D1,D2        ;D2=*c=*c+b=25
         MOVE L      D2,(A0)      ;k=*c is pushed back
         RTS                      ;Return from subroutine
```

Fig. 1 Describes how the C compiler pushes arguments onto the MC68000 stack

Example 2: In this example, the following equations which are used in 3-D image transformation and animation are, implemented in MC68000 assembly language. The way, the C compiler pushes the arguments onto the MC68000 stack is shown in Fig. 2. The relevant memory map which clearly displays how the coordinates of each vertex is stored in the memory is also presented. The memory map presented in Fig. 2, is essential and very helpful during the implementation process.

$$x' = x \qquad \ldots(1)$$
$$y' = y\cos A - z\sin A \qquad \ldots(2)$$
$$z' = y\sin A + z\cos A \qquad \ldots(3)$$

Where x', y', z' are the newly modefied values of x, y, and z respectivelly.



```
#include <stdio.h>
extern rotx( );
typedef struct vertex_rec {
                    int    x;
                    int    y;
                    int    z;
} VERTEX;
main( )
{
  int  size, sinA, cosA;
  VERTEX eye[8];
  eye[0].x = 0;   eye[0].y = 0;   eye[0].z = 0;
  eye[1].x = 0;   eye[1].y = 5;   eye[1].z = 0;
  eye[2].x = 5;   eye[2].y = 5;   eye[2].z = 0;
  eye[3].x = 5;   eye[3].y = 0;   eye[3].z = 0;
  eye[4].x = 5;   eye[4].y = 0;   eye[4].z = 5;
  eye[5].x = 0;   eye[5].y = 0;   eye[5].z = 5;
  eye[6].x = 0;   eye[6].y = 5;   eye[6].z = 5;
  eye[7].x = 5;   eye[7].y = 5;   eye[7].z = 5;
  sinA = 1;
  cosA = 0;
  size = 8;
  rotx(&eye[0], size, sinA, cosA);
}
```

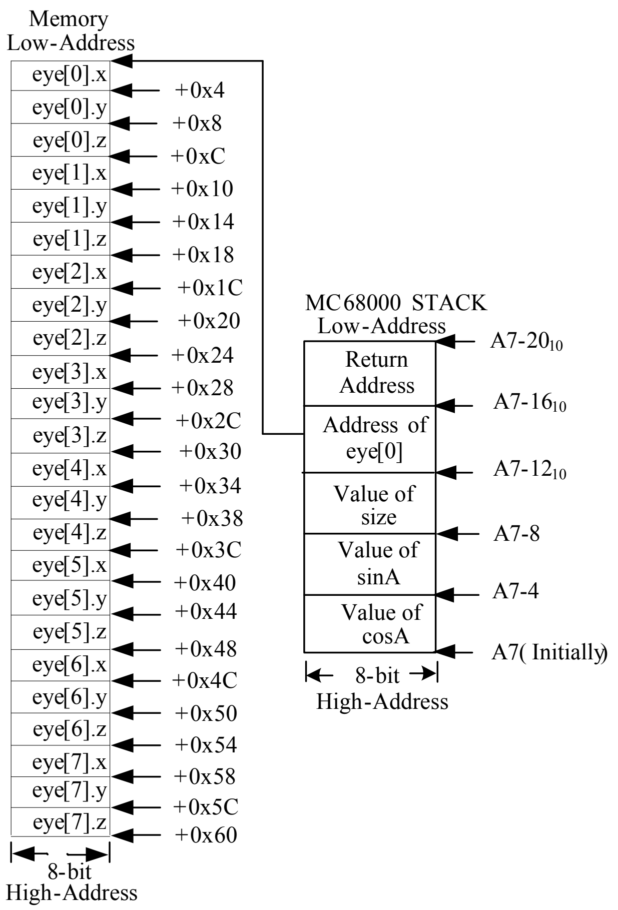


Fig. 2a Describes how the C compiler pushes arguments onto MC68000 stack and the corresponding memory-map

```
        opt         case
        section     code,,c
        xdef        _rotx

rotx:   MOVE.L   4(A7),A0      ;A0=&eye[0]
        MOVE.L   8(A7),D0      ;D0=size=8
        MOVE.L   12(A7),D1     ;D1=sinA
        MOVE.L   16(A7),D2     ;D2=cosA
        SUB.L    #1,D0         ;Counter

loopx:  MOVE.L   4(A0),D3      ;D3=eye[i].y
        MOVE.L   8(A0),D4      ;D4=eye[i].z
        MOVE.L   D3,D5         ;D5=copy
        MOVE.L   D4,D6         ;D6=copy
        MULU     D2,D3         ;D3=ycosA
        MULU     D1,D4         ;D4=zsinA
        SUB.L    D4,D3         ;D3= y'
        ADD.L    #4,A0         ;Offset
        MOVE.L   D3,(A0)+      ;y' is pushed

        MULU     D1,D5         ;D5=ysinA
        MULU     D2,D6         ;D6=zcosA
        ADD.L    D5,D6         ;D6=z'
        MOVE.L   D6,(A0)+      ;z' is pushed

        DBRA     D0,loopx      ;Counter
        RTS                    ;Return
```

Fig. 2b Implementation of the C function **rotx( )** in MC68000 assembly language

III. Interfacing **C** and TMS320C6713 Assembly

There are three **ways** in which the C compiler passes arguments from one function to another in the TMS320C6713-based environment. In the case of pure C programming, the user of the TMS320C6713-based system does not need to know and also does not need to worry how the C compiler passes the arguments from one function to another. However, in the case of interfacing C with TMS320C6713 assembly language subroutine, it is crucially important for a user to understand how the C compiler passes the arguments from a C function into a TMS320C6713 assembly subroutine. In the following sections, only one **way** is presented in detail. Other **ways** will be presented in the future.

A. Passing Arguments through the Registers Only

In this case, the C compiler places the arguments inside the registers in a special manner and the user of the TMS320C6713-based system needs to be aware of this fact and use it correctly when interfacing C with the TMS320C6713 assembly language [3], [8].

It is vitally important for a user of the TMS320C6713-based system to properly understand and follow the register conventions when interfacing C with TMS320C6713 assembly language subroutine. The register conventions dictate how the C compiler uses registers for passing arguments between functions and how values are preserved across function calls [3], [5].

When a calling function passes arguments to a called function, up to the first 10 arguments are placed in registers A4, B4, A6, B6, A8, B8, A10, B10, A12, and B12 respectively and the remaining arguments are placed on the stack [3]. As shown in **example-3**, the integer values of the arguments **i** and **j** are placed in registers A4 and B4 respectively; while the address of the argument **k** is placed in register A6 (see Fig. 3a). By convention, the first argument is the left most argument (i.e. ***i*** in this case). For better understanding, the C function (**asmfunc**) is converted into the TMS320C6713 assembly language subroutine (see Fig. 3b). The return address to the calling function is normally placed in B3 and for this reason a branch to B3 needs to be performed at the end of the assembly language subroutine. It is worth mentioning that the way the C compiler passes arguments from the calling function to the called function in the TMS320C6713-based environment is totally different from the way the C compiler passes arguments in a conventional microprocessor such as MC68000 [1]-[2]. It should be noted that this example gives the same correct result when the TMS320C6713 DSK board is operated either in little-endian or in big-endian mode.

In example-4, the address of the first argument is placed in register A4 and the integer value of the second argument is placed in B4; while the floating-point values of the third and fourth arguments are placed in registers A6 and B6 respectively (see Fig. 4a). This example presents the implementation of the equations used in 3-D image transformation and animation, in TMS320C6713 assembly language. The C function (**asmfunc**) is converted into the TMS320C6713 assembly language subroutine as shown in Fig. 4b. It should be noted that this example works correctly and produces the same correct result in both little-endian and big-endian mode of the TMS320C6713 DSK board (i.e. endianness in this example does not really matter).

Example-5 demonstrates how the C compiler places the floating-point values of the arguments **x** and **y** in registers A4 and B4 respectively and places the address of the argument **z** in register A6 as shown in Fig. 5a. Appropriate TMS320C6713 assembly language instructions such as single-precision are used for floating-point data manipulation. The conversion of the C function (**asmfunc**) into the TMS320C6713 assembly language is presented in Fig. 5b. It should be noted that in this example, the endianness of the TMS320C6713 DSK board also does not matter.

In example-6, the double-precision values of the arguments **x** and **y** are placed in register pairs **A5:A4** and **B5:B4** respectively; while the address of the argument **z** is placed in register **A6** (see Fig. 6a). In other words, the upper and lower 32-bits of **x** and **y** are placed in register pairs A5:A4 and B5:B4 respectively. The double-precision value of **z** itself is stored as 64-bits in memory as shown in Fig. 6b by the memory layout. Appropriate assembly language instructions such as double-precision addition (ADDDP) and double-precision load (LDDW) are employed for data manipulation [6]-[7]. The reader needs to pay attention to the way the final double-precision value of **z** is stored into the memory when the

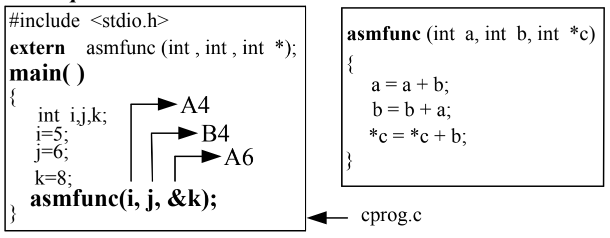
TMS320C6713 board is operated in the little-endian mode (see Fig. 6b).

In example-7, the double-precision values of the arguments **x** and **y** are placed in register pairs **A5:A4** and **B5:B4** respectively; while the address of the argument **z** is placed in register **A6**. In other words, the upper and lower 32-bits of **x** and **y** are placed in register pairs A5:A4 and B5:B4 respectively (see Fig. 7a). The double-precision value of **z** itself is stored as 64-bits in memory as shown in Fig. 7b by the memory layout. Appropriate assembly language instructions such as double-precision addition (ADDDP) and double-precision load (LDDW) are employed for data manipulation [6]-[7]. The reader needs to pay attention to the way the final double-precision value of **z** is stored into the memory when the TMS320C6713 board is operated in big-endian mode. Thorough comparison of examples 6 and 7 will clarify the difference using the two modes of the board.

In **example-8**, the long values of the arguments **x** and **y** are placed in register pairs A5:A4 and B5:B4 respectively; and the address of the argument **z** is placed in register A6. In other words, the upper and lower 32-bits of **x** and **y** are placed in register pairs A5:A4 and B5:B4 respectively (see Fig. 8a). The long value of **z** itself is stored as 64-bits as shown in the memory-layout. Appropriate assembly language instructions are employed for data manipulation. The reader is encouraged to pay lots of attention to the implementation of the C function (asmfunc) into the TMS320C6713 assembly language as shown in Fig. 8b, especially to the way the final long value of **z** is stored into the memory in little-endian.

Finally, in **example-9**, the long values of the arguments **x** and **y** are placed in register pairs A5:A4 and B5:B4 respectively; and the address of the argument **z** is placed in register A6. In other words, the upper and lower 32-bits of **x** and **y** are placed in register pairs A5:A4 and B5:B4 respectively (Fig. 9a). The long value of **z** itself is stored as 64-bits as shown in the memory-layout. Appropriate assembly language instructions are employed for data manipulation. The reader is encouraged to pay lots of attention to the implementation of the C function (asmfunc) into the TMS320C6713 assembly language as shown in Fig. 9b, especially to the way the final long value of **z** is stored into the memory in big-endian mode. Thorough comparison of examples 8 and 9 will highlight the difference.

Example 3



```
#include <stdio.h>
extern  asmfunc (int , int , int *);
main( )
{
  int  i,j,k;
  i=5;
  j=6;
  k=8;
  asmfunc(i, j, &k);
}
```

```
asmfunc (int  a, int  b, int  *c)
{
  a = a + b;
  b = b + a;
  *c = *c + b;
}
```

Fig. 3a Presents how C compiler places arguments into TMS320C6713 registers

```
           global     _asmfunc
           text

_asmfunc   ADD L1X    A4,B4,A4      ;A4=i=a =a+b=5+6=11
           NOP        5             ;5 delay slots
           ADD L2X    B4,A4,B4      ;B4=j=b=b+a=17
           NOP        5
           LDW        *A6,A8        ;A8=k=*c=8
           NOP        5
           ADD L1X    A8,B4,A8      ;A8=k=*c+b=25
           NOP        5
           STW        A8,*A6        ;The value of k=*c is stored
           NOP        5
           B          B3            ;return to the calling function
           NOP        5             ;5 delay slots for branch
```

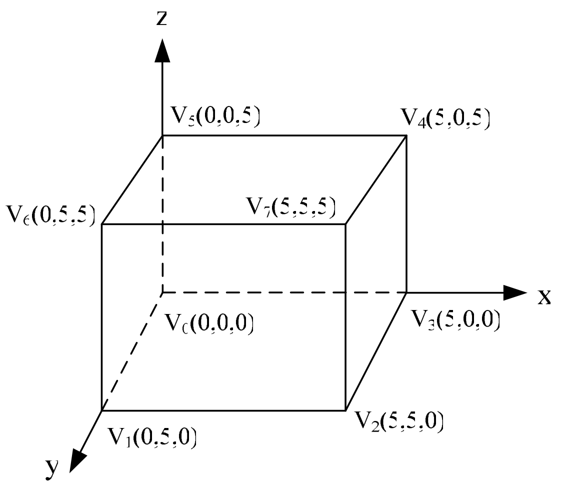
assembly asm

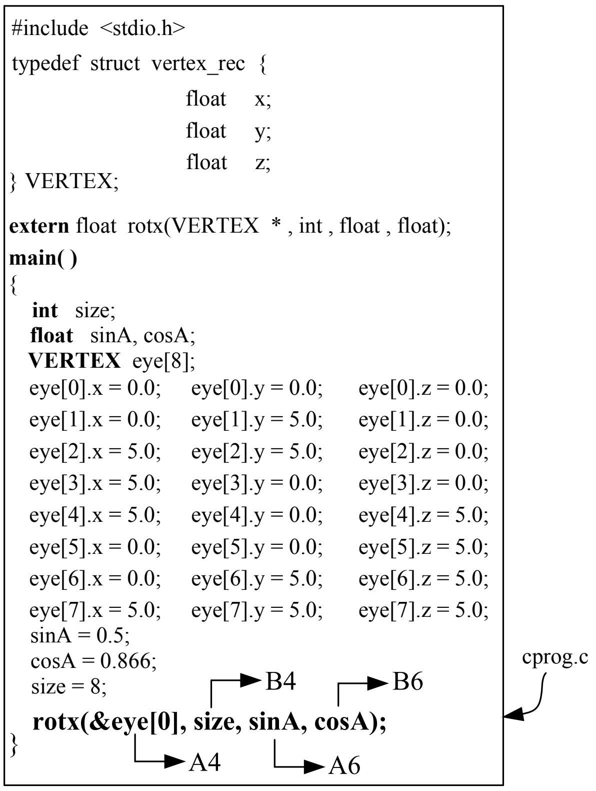
Fig. 3b Implementation of the C function **asmfun( )** in TMS320C6713 assembly language

Example 4: In this example, the following equations which are used in 3-D image transformation and animation are, implemented in TMS320C6713 assembly language. The way the C compiler passes the arguments from the calling function in C to a called function in TMS320C6713 assembly is displayed. The memory map, which is crucially important during the implementation process, also presented.

$x' = x$	..(1)
$y' = y\cos A - z\sin A$	..(2)
$z' = y\sin A + z\cos A$	..(3)

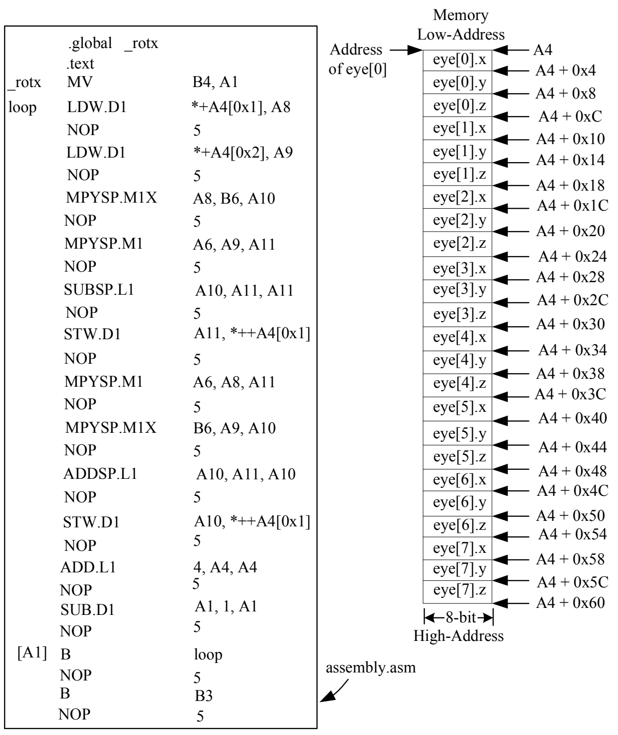
Where x' , y', z' are the newly modefied values of x, y, and z respectivelly.





```
#include  <stdio.h>
typedef struct  vertex_rec  {
                        float    x;
                        float    y;
                        float    z;
} VERTEX;

extern float  rotx(VERTEX  * , int , float , float);
main( )
{
   int   size;
   float  sinA, cosA;
   VERTEX  eye[8];
   eye[0].x = 0.0;    eye[0].y = 0.0;    eye[0].z = 0.0;
   eye[1].x = 0.0;    eye[1].y = 5.0;    eye[1].z = 0.0;
   eye[2].x = 5.0;    eye[2].y = 5.0;    eye[2].z = 0.0;
   eye[3].x = 5.0;    eye[3].y = 0.0;    eye[3].z = 0.0;
   eye[4].x = 5.0;    eye[4].y = 0.0;    eye[4].z = 5.0;
   eye[5].x = 0.0;    eye[5].y = 0.0;    eye[5].z = 5.0;
   eye[6].x = 0.0;    eye[6].y = 5.0;    eye[6].z = 5.0;
   eye[7].x = 5.0;    eye[7].y = 5.0;    eye[7].z = 5.0;
   sinA = 0.5;
   cosA = 0.866;
   size = 8;
   rotx(&eye[0], size, sinA, cosA);
}
```

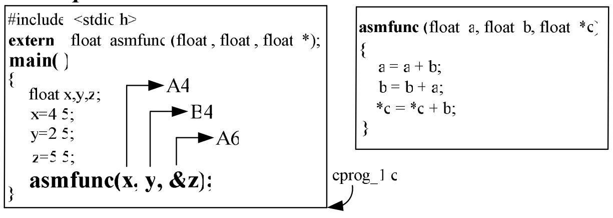
Fig. 4a Presents how C compiler places arguments into TMS320C6713 registers



```
        .global   _rotx
        .text
_rotx   MV              B4, A1
loop    LDW.D1          *+A4[0x1], A8
        NOP             5
        LDW.D1          *+A4[0x2], A9
        NOP             5
        MPYSP.M1X       A8, B6, A10
        NOP             5
        MPYSP.M1        A6, A9, A11
        NOP             5
        SUBSP.L1        A10, A11, A11
        NOP             5
        STW.D1          A11, *++A4[0x1]
        NOP             5
        MPYSP.M1        A6, A8, A11
        NOP             5
        MPYSP.M1X       B6, A9, A10
        NOP             5
        ADDSP.L1        A10, A11, A10
        NOP             5
        STW.D1          A10, *++A4[0x1]
        NOP             5
        ADD.L1          4, A4, A4
        NOP             5
        SUB.D1          A1, 1, A1
        NOP             5
[A1]    B               loop
        NOP             5
        B               B3
        NOP             5
```

Fig. 4b The memory map and the translation of the function, **rotx( )** into the TMS320C6713 assembly language

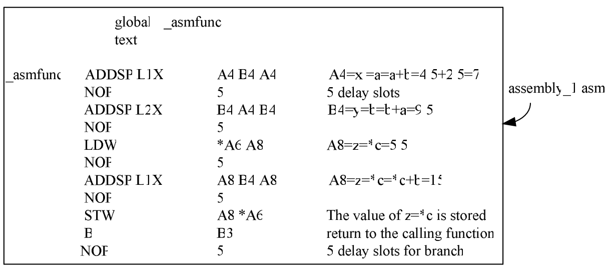
Example 5



```
#include  <stdio.h>
extern   float  asmfunc (float , float , float *);
main( )
{
   float x,y,z;
   x=4.5;
   y=2.5;
   z=5.5;
   asmfunc(x, y, &z);
}
```

```
asmfunc (float  a, float  b, float  *c)
{
   a = a + b;
   b = b + a;
   *c = *c + b;
}
```

Fig. 5a How C compiler places arguments into C6713 registers

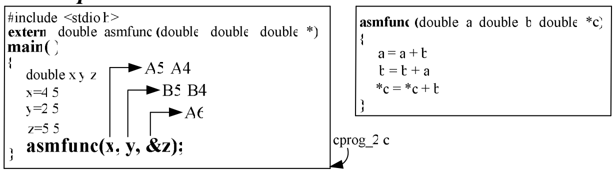


```
              global    _asmfunc
              text

_asmfunc   ADDSP L1X       A4 B4 A4         A4=x =a=a+b=4 5+2 5=7
           NOP             5                5 delay slots
           ADDSP L2X       B4 A4 B4         B4=y=b=b+a=9 5
           NOP             5
           LDW             *A6 A8           A8=z=*c=5 5
           NOP             5
           ADDSP L1X       A8 B4 A8         A8=z=*c=*c+b=15
           NOP             5
           STW             A8 *A6           The value of z=*c is stored
           B               B3               return to the calling function
           NOP             5                5 delay slots for branch
```

assembly_1 asm

Fig. 5b Translation of **asmfunc( )** into TMS320C6713 assembly language subroutine

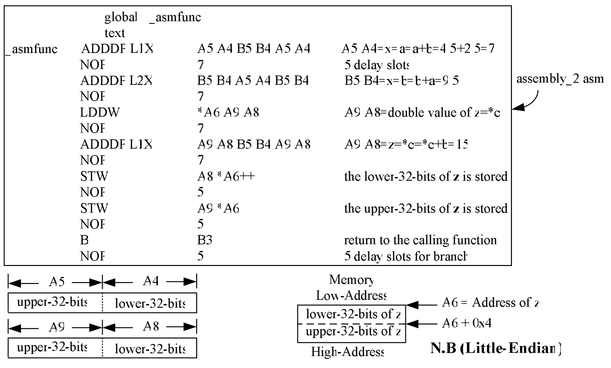
Example 6



```
#include <stdio h>
extern  double  asmfunc (double   double   double  *)
main( )
{
   double x y  z
   x=4 5
   y=2 5
   z=5 5
   asmfunc(x, y, &z);
}
```

```
asmfunc (double  a  double  b  double  *c)
{
   a = a + b
   b = b + a
   *c = *c + b
}
```

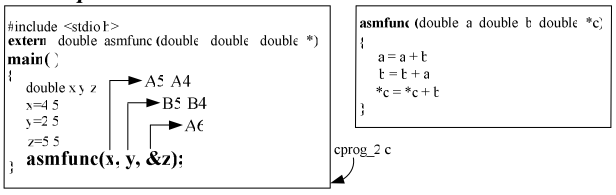
Fig. 6a How C compiler places arguments into C6713 registers



```
              global    _asmfunc
              text
_asmfunc   ADDDP L1X     A5 A4 B5 B4 A5 A4     A5 A4=x=a=a+b=4 5+2 5=7
           NOP           7                     5 delay slots
           ADDDP L2X     B5 B4 A5 A4 B5 B4     B5 B4=x=b=b+a=9 5
           NOP           7
           LDDW          *A6 A9 A8             A9 A8=double value of z=*c
           NOP           7
           ADDDP L1X     A9 A8 B5 B4 A9 A8     A9 A8=z=*c=*c+b=15
           NOP           7
           STW           A8 *A6++              the lower-32-bits of z is stored
           NOP           5
           STW           A9 *A6                the upper-32-bits of z is stored
           NOP           5
           B             B3                    return to the calling function
           NOP           5                     5 delay slots for branch
```

Fig. 6b Memory-map and conversion of **asmfunc( )** into C6713 assembly language subroutine in little-endian

Example 7



```
#include <stdio h>
extern  double  asmfunc (double   double   double  *)
main( )
{
   double x y  z
   x=4 5
   y=2 5
   z=5 5
   asmfunc(x, y, &z);
}
```

```
asmfunc (double  a  double  b  double  *c)
{
   a = a + b
   b = b + a
   *c = *c + b
}
```

Fig. 7a How C compiler places arguments in C6713 registers

```
                global   _asmfunc
                text
_asmfunc    ADDDP L1X     A5 A4 B5 B4 A5 A4     A5 A4=x=a=a+b=4 5+2 5=7
            NOP           7                     5 delay slots
            ADDDP L2X     B5 B4 A5 A4 B5 B4     B5 B4=x=b=b+a=9 5
            NOP           7
            LDDW          *A6 A9 A8             A9 A8=double value of z=*c
            NOP           7
            ADDDP L1X     A9 A8 B5 B4 A9 A8     A9 A8=z=*c=*c+b=15
            NOP           7
            STW           A9 *A6++              the upper-32-bits of z is stored
            NOP           5
            STW           A8 *A6                the lower-32-bits of z is stored
            NOP           5
            B             B3                    return to the calling function
            NOP           5                     5 delay slots for branch
```
assembly_3 asm

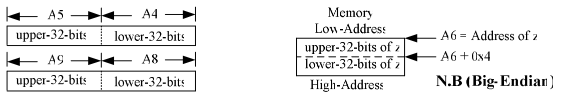
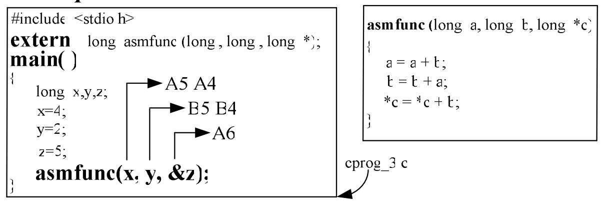


Fig. 7b Memory-map and conversion of **asmfunc( )** into C6713 assembly language subroutine in big-endian

Example 8



```
#include <stdio h>
extern long asmfunc (long , long , long *);
main( )
{
  long x,y,z;
  x=4;
  y=2;
  z=5;
  asmfunc(x, y, &z);
}
```

```
asmfunc (long a, long b, long *c)
{
  a = a + b;
  b = b + a;
  *c = *c + b;
}
```

Fig. 8a Describes how C compiler places arguments into TMS320C6713 registers

```
                global   _asmfunc

                text
_asmfunc    ADD L1X     B4 A5 A4 A5 A4     A5 A4=x=a=a+b=4+2 5=6
            NOP         7                  7 delay slots
            ADD S1X     B5 A5 A5           A5 A4 is now completed
            NOP         5
            ADD L2X     A4 B5 B4 B5 B4     B5 B4=y=b=b+a=8
            NOP         7
            ADD S2X     A5 B5 B5           B5 B4 is now completed
            NOP         5
            LDDW        *A6 A9 A8          A9 A8=long value of z=*c
            NOP         7
            ADD L1X     B4 A9 A8 A9 A8     A9 A8=z=*c=*c+b=13
            NOP         7
            ADD S1X     B5 A9 A9           A9 A8 is now completed
            NOP         5
            STW         A8 *A6++           the lower-32-bits of z is stored
            NOP         5
            STW         A9 *A6             the upper-32-bits of z is stored
            NOP         5
            B           B3                 return to the calling function
            NOP         5                  5 delay slots for branch
```
assembly_4 asm

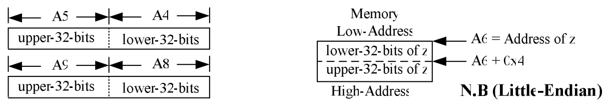
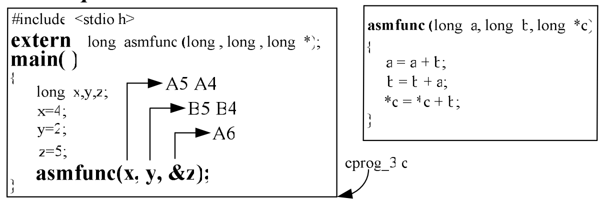


Fig. 8b Memory-map and conversion of **asmfunc( )** into C6713 assembly language subroutine in little-endian.

Example 9



```
#include <stdio h>
extern long asmfunc (long , long , long *);
main( )
{
  long x,y,z;
  x=4;
  y=2;
  z=5;
  asmfunc(x, y, &z);
}
```

```
asmfunc (long a, long b, long *c)
{
  a = a + b;
  b = b + a;
  *c = *c + b;
}
```

Fig. 9a Presents how C compiler places arguments into TMS320C6713 registers

```
                global   _asmfunc

                text
_asmfunc    ADD L1X     B4 A5 A4 A5 A4     A5 A4=x=a=a+b=4+2 5=6
            NOP         7                  7 delay slots
            ADD S1X     B5 A5 A5           A5 A4 is now completed
            NOP         5
            ADD L2X     A4 B5 B4 B5 B4     B5 B4=y=b=b+a=8
            NOP         7
            ADD S2X     A5 B5 B5           B5 B4 is now completed
            NOP         5
            LDDW        *A6 A9 A8          A9 A8=long value of z=*c
            NOP         7
            ADD L1X     B4 A9 A8 A9 A8     A9 A8=z=*c=*c+b=13
            NOP         7
            ADD S1X     B5 A9 A9           A9 A8 is now completed
            NOP         5
            STW         A9 *A6++           the upper-32-bits of z is stored
            NOP         5
            STW         A8 *A6             the lower-32-bits of z is stored
            NOP         5
            B           B3                 return to the calling function
            NOP         5                  5 delay slots for branch
```
assembly_5 asm

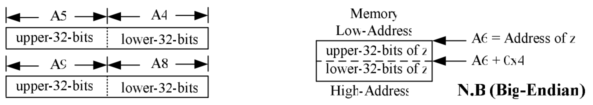


Fig. 9b Memory-map and conversion of **asmfunc( )** into C6713 assembly language subroutine in big-endian

IV. Conclusion

One method of interfacing C with the TMS320C6713 assembly language has been comprehensively described. The concept presented in this paper will be essential and of great interest to many users who are employing a micro-based system for their applications; and especially for those users who want to use the TMS320C6713-based system for assembly language programming and signal processing. It is strongly recommended to the users of the TMS320C6713-based systems to properly understand and follow the register conventions when interfacing C with the TMS320C6713 assembly language subroutine; otherwise, it would cause lots of confusion and erroneous debugging results as far as passing arguments among various functions is concerned. The presented software and concept have been tested extensively by examining different types of examples under various conditions and has proved highly reliable in operation. Finally, the user in some cases needs to take into consideration the endianness of the TMS320C6713 DSK board during the interfacing of C with the TMS320C6713 assembly language (see examples 6-9).

Acknowledgment

I would like to thank Southampton Solent University, Faculty of Technology, for giving me the opportunity and help to carry out this work. I would also like to thank Kevin Walsh and Jomo Batola for their support and encouragement.

References

[1] A A Wardak, G A King, R Backhouse, Interfacing high-level and assembly language with microcodes in 3-D image generation. Journal of Microprocessors and Microsystems, Vol. 18, No.4, May 1994, Butterworth-Heinemann Ltd.
[2] A A Wardak, Real-Time 3-D Image Generation with TMS320C30 EVM, Journal of Microcomputer Applications, Vol. 18, pp 355-373, 1995, Academic Press Limited.
[3] TMS320C6000 Optimizing C Compiler User's Guide, SPRU187K, Texas Instruments, Dallas, TX, 2002.
[4] R Chassaing, Digital Signal Processing and Applications with the 6713 and C6416 DSK, Wiley, New York, 2005.

[5] TMS320C6000 Programmer's Guide, SPRU198G, Texas Instruments, Dallas, TX, 2002.
[6] TMS320C6000 CPU and Instruction Set Reference Guide, October 2000, Literature Number: SPRU189F.
[7] TMS320C6000 Assembly Language Tools, User's Guide, Literature Number: SPRU186K, October 2002.
[8] http://193.140.141.8/~redizkan/Interfacing_C_and_Assembly.pdf

Dr. Abdullah Wardak was a lecturer at the Kabul University, Faculty of Engineering, Kabul, Afghanistan from 1978 to 1983, where he received his BSc in electrical and electronics engineering in 1978. He received his MSc in electronic control engineering in 1987 from Salford University, UK. He received his PhD in 1991 from Bradford University, UK. Currently he is a senior lecturer in Southampton Solent University, Faculty of Technology, School of Computing and Communications.