Application Report
SPRA493

How to Interface C and Assembly Language with the TMS320 Floating Point C Compiler

Yves Gagniere and Dirk Langmesser

Digital Signal Processing Solutions

Abstract

The Texas Instruments (TI™) TMS320 digital signal processor (DSP) floating-point ANSI C compiler generally uses a conventional mechanism for passing arguments and allocating local variables. Nevertheless, the programmer has several options to pass arguments to a function, to allocate automatic variables and to return a value from a called function.

If the programmer uses only the C language, the C compiler manages a dedicated mechanism depending on the selected options at compile time. The mechanism regulates the data flow between the calling and the called function. These strict conventions are transparent to the user. If the user plans to interface assembly functions or routines to a C program, it is straightforward to follow the calling and returning conventions as well as the calling and called function's responsibilities regarding the saving and restoring of registers.

This application report offers a simple approach using case studies to describe the main choices available to the programmer to interface C and assembly language. This report is based on revision 5 of the Texas Instruments TMS320 floating-point tools.

Contents

The Memory Model

The TMS320 floating-point family supports two different memory models that affect how the global and static variables are accessed. Generally, direct addressing of a data word in memory is realized through paging by the concatenation of the 8 (TMS320C3x) or 16 (TMS320C4x) least significant bits of the data page pointer (DP) with the 16 least significant bits of the instruction word. From that, the size of data pages on the C3x or C4x is 64K-words. The number of pages is 256 on the C3x and 64k on the C4x, respectively.

The main point is that in the small memory model the complete .bss section, which is the standard C-section for global and static variables, will be accessed without ever changing the DP pointer. This means evidently that program space and cycle count might be saved through this method.

With the small memory model, only the 16 bits of the instruction word can be used for addressing. From that, there is the requirement that the entire collection of global and static variables fit within one page, which is a 64K-word contiguous memory space that should be aligned on a 64K-word boundary – this means that the .bss section has to be linked appropriately. The small memory option is the default and forces the compiler to initialize the data page pointer during runtime initialization to point to the beginning of the .bss section. Now the C compiler can access all objects located in the .bss section using direct addressing without ever modifying the data page pointer.

On the other hand, the big memory model does not limit the size of the .bss section and therefore the number of global and static variables – only the memory space available on the target system causes a restriction. The C compiler accesses all of the objects located in the .bss section with direct addressing, modifying the data page pointer before each access and causing at least one additional operation cycle. The big memory model option can be selected with the -mb command-line switch.

Observe that large globally declared static array objects such as

int table[100000]

will force you to use the big memory model. A more efficient implementation allocates such an array dynamically:

```
int *table;                                      /* global declaration          */
table = (int *) malloc (100000 * sizeof(int));   /* allocation in main program  */
```

The data object is then allocated on the heap. The size of the heap is determined in the linker command file with the -heap option. The heap is identical to the .sysmem-section, which can be located in the memory system also through the linker command file. The access to heap elements is realized in an indirect way with pointer addressing.

The RAM/ROM Model

C Code should generally be linked with either the -c or the -cr option. The C compiler then produces code linked with a standard library function called boot.asm located in the runtime support library (rts.lib). Depending on the option chosen at link time, the initialization of variables will be performed by the boot function at either load time (-cr: RAM model) or runtime (-c: ROM model).

If the RAM model is selected, it will be the responsibility of a smart loader to initialize the variables in memory (built-in to TI simulator and debugger). If the ROM model is selected, the .cinit section is used to store the initialization data of global and static variables in ROM, which will be copied automatically by the C boot routine to the appropriate locations within .bss before entering the C main routine. If C is interfaced with assembly code that uses the .bss section, it will be necessary to create a .cinit table and a .bss section entry for the assembly code according to the C conventions.

The .bss and .cinit definitions should have the following format in assembly to be used by a loader or the C boot code routine:

```
.bss     _variablename, size      ;  _variablename is the reference name for the data structure
                                  ;  used in the C code; size is the number of 32 bit words;
.sect    ".cinit"                 ;  section that contains tables with the values for initializing
                                  ;  the variables and constants;
.word    size                     ;  size is the number of 32 bits words for the data structure;
.word     _variablename           ;  _variablename is the reference name for the data structure;
.word    value0, value1,...       ;  values which will be assigned to the
                                  ; _variablename structure;
```

Function Call

The C compiler imposes a strict set of rules on function calls. Failure to adhere to these rules can disrupt the C environment and cause the program to fail. The programmer can choose between two runtime models for function calls. The first model, the *stack argument runtime model*, intensively uses the stack to pass arguments between the calling and the called function. The called function additionally uses the stack for its local frame and temporary variables as well as to pass arguments as soon as it becomes a calling function on its own.

The second model is called the *register argument runtime model* and uses the register set to pass arguments from the calling to the called function. The stack passing convention applies to those arguments for which not enough registers are available to pass them all and those that are not compatible with the register use definition.

The stack argument runtime model is the default mode. The more efficient register argument runtime model can be activated with the -mr command-line switch of the compiler.

Preserving Registers

The C language is based on function calls, where the first function called from the C boot routine is the “main” function. Entering an assembly context from a C routine makes it necessary to preserve the C environment. Basically this means that those registers used in the C environment must be saved across calls: these registers are called *register variables.*

It is the called function’s responsibility to preserve the contents of these registers if they are used inside the function. This type of preserving scheme is commonly called *save-on-entry*. The C compiler can use the remaining registers to evaluate expressions and store temporary results. This strategy allows the C compiler to reuse register data, take advantage of the efficient register addressing modes, and avoid unnecessary accesses to the local frame. The contents of these registers are not preserved across calls and therefore must be saved on the local frame of the caller function.

The save-on-entry registers (AR3, SP, R4, R5, R6, R7, AR4-AR7, and R8 (C4x only)) are saved by the called function after entry provided they are used inside the function. The optimizer tries to allocate all variables to registers – if the optimizer is not used, the compiler can be forced through the register keyword to try to assign one of the save-on-entry registers to a variable.

The following table summarizes the usage of the registers and highlights the save-on-entry registers:

Register	**Use without Optimizer**	**Use with Optimizer**	**Called Function Responsibility or Save on Entry**
R0	Integer and float expressions or scalar return values	Integer and float expressions or scalar return values	no
R1	Integer and float expressions	Integer and float expressions	no
R2-R3	Integer and float expressions	Integer and float register variables	no
R4-R5	Integer register variables	Integer or float register variables	integer part
R6-R7	Float register variables	Integer or float register variables	floating part
AR0-AR1	Pointer expressions	Pointer expressions	no
AR2	Pointer expressions	Integer and pointer register variables	no
AR3	Frame pointer (FP)	Frame pointer (FP)	yes
AR4-AR7	Pointer register variables	Pointer register variables	yes
IR0	Extended frame offsets	Extended frame offsets	no
IR1	Extended frame offsets	Integer register variables	no
BK	Not used	Integer register variables	no
RC,RS	Block (structure) copy	Block (structure) copy	no
RE	Not used	Block repeat loop or integer register variables	no
SP	Stack pointer	Stack pointer	yes
DP	Accessing global variables (big model only)	Accessing global variables (big model only)	yes/no**
R8*	Integer register variables	Integer and float register variables	integer part
R9-R11*	Integer and float variables	Integer and float variables	no

* C4x only
** Yes in small model/no in big model

In the special case of the implementation of an interrupt service routine, all registers used inside the routine must be saved on the stack. In C, you can use the interrupt pragma to program an interrupt service routine – the necessary preserving of registers will then be done automatically.

The Stack Argument Runtime Model

The TMS320 floating point C compiler uses a conventional mechanism to manage the stack for passing arguments and allocating space for the local variables of a function. The stack pointer is a hardware register. The stack grows toward higher addresses and the stack pointer always points to the last element pushed onto the stack, i.e., the top of the stack. The size of the stack is determined through the linker command file with the -stack option.

A second register called Frame Pointer (FP) points to the local frame of the called function. Basically, this pointer is the reference address to access both the arguments and the local frame. It has to be seen that the local frame is accessed in an indirect manner with pointer and offset addressing. From that, a restriction is caused by the maximum offset size of eight bits. Consequently, the number of local variables that can be accessed without overhead is also restricted: the local frame should not excess 256 words. If this restriction cannot be met, the most often used variables should be declared first. An access to the local frame across the 256 word boundary causes one extra cycle for loading the offset to an index register plus two additional pipeline delay cycles.

The FP has been defined with a syntactic replacement for readability purposes: *FP .set AR3*. This means that the FP is not a processor register but that AR3 is used by convention for this purpose and FP is the AR3 nickname for the C compiler. The following C code “example0.c” is shown here as a learning exercise.

```
int func(int e, int f);     /* prototype of func;                           */
int c=3,d=4;                /* c & d are global variables;                  */
void main(void)             /* first function called;                       */
{
int a=1,b=2;                /* automatic or local variables;                */
for (;;)                    /* infinite loop;                               */
             {
             d = d + 1;     /* increment the global variable;               */
             a = func(c,d); /* call a function that has  two arguments      */
                            /*  and that should return an integer;          */
             b = a + 2;     /* sum up the local variable;                      */
             }
}

int func (int e, int f)     /* function that has two arguments and that        */
{                           /* returns an integer as result;                   */
int x;                      /* this function has one local variable;        */
x = e + f;                     /* sum of parameters;                           */
x = x + 2;                  /* sum of automatic variable and constant;      */
return(x);                  /* return result to caller;                        */
}
```

The C compiler call cl30 -k example0.c produces the following assembly code. The options are:

❒ Stack argument runtime model

❒ Small memory model

❒ No optimization

From that, some parts of the code could be implemented more efficiently; however, the readability would be worse.

```
;******************************************************************************
;* TMS320C3x/4x ANSI C Code Generator                    Version 5.00          *
;* Date/Time created: Thu Dec 11 14:43:05 1997                                 *
;******************************************************************************
         .regalias ; enable floating point register aliases
fp       .set        ar3
FP       .set        ar3
;******************************************************************************
;* GLOBAL FILE PARAMETERS                                                      *
;*                                                                          *
;*   Optimization   : Always Choose Smaller Code Size                       *
;*   Memory         : Small Memory Model                                    *
;*   Float-to-Int   : Normal Conversions (round toward -inf)                *
;*   Multiply       : in Software (32 bits)                                 *
;*   Memory Info    : Unmapped Memory Exists                                *
;*   Repeat Loops   : Use RPTS and/or RPTB                                  *
;*   Calls          : Normal Library ASM call                               *
;*   Debug Info     : No Debug Info                                         *
;****************************************************************************
;       C:\C3xTOOLS\ac30.exe example0.c C:\temp\example0.if

        .sect       ".cinit"            ; definitions of .cinit tables
        .field      1,32
        .field      _c+0,32
        .field      3,32                ; _c @ 0

        .sect       ".text"

        .global     _c
        .bss        _c,1                ; reservation of un-initialized space for global variable c

        .sect       ".cinit"
        .field      1,32
        .field      _d+0,32
        .field      4,32                ; _d @ 0

        .sect       ".text"

        .global     _d
        .bss        _d,1                ; reservation of un-initialized space for global variable d
        .sect       ".text"

        .global     _main
;***************************************************************************
;* FUNCTION NAME: _main                                                    *
;*                                                                         *
;*   Architecture       : TMS320C30                                        *
;*   Calling Convention : Stack Parameter Convention                       *
;*   Function Uses Regs : r0,r1,sp                                         *
;*   Regs Saved         :                                                  *
;*   Stack Frame        : Full (Frame Pointer in AR3)                      *
;*   Total Frame Size   : 2 Call + 0 Parm + 2 Auto + 0 SOE = 4 words       *
;***************************************************************************
_main:                                  ; step 1   is performed by call _main in the C boot routine
     push       fp                      ; step 2   save old fp in local frame
     ldiu       sp,fp                   ;          initialize new fp
     ldiu       1,r1
     addi       2,sp                    ; step 3   allocate local frame for _a and _b
     ldiu       2,r0
     sti        r1,*+fp(1)              ; step 3   initialize _a in local frame
     sti        r0,*+fp(2)              ;          initialize _b in local frame
L2:
     ldiu       1,r0
     addi       @_d+0,r0                ;          access _d (defined in .bss) with direct addressing and add 1
     sti        r0,@_d+0                ;          write back _d + 1
     push       r0                      ; step 4   revised _d  is pushed on the stack
```

```
      ldiu     @_c+0,r0            ;           access _c with direct addressing
      push     r0                  ; step 4    _c is pushed on the stack
      call     _func               ; step 5    call the _func function and save the return address on stack
                                   ; Call Occurs
      ldiu     2,r1
      sti      r0,*+fp(1)          ;           places the return value (r0) to _a location
      bud      L2                  ;           branch delayed to beginning of for(;;) loop
      addi     *+fp(1),r1          ;           _b is computed
      sti      r1,*+fp(2)          ;           the result is stored in the _b location
      subi     2,sp                ; step 9    pops the arguments _d and _c
                                   ;  Branch Occurs to L2
                                   ; A return to the C boot routine does not occur here since the main
                                   ; program is looping endlessly


         .sect      ".text"

         .global    _func
;******************************************************************************
;* FUNCTION NAME: _func                                                       *
;*                                                                            *
;*   Architecture       : TMS320C30                                           *
;*   Calling Convention : Stack Parameter Convention                          *
;*   Function Uses Regs : r0,r1                                              .*
;*   Regs Saved         :                                                     *
;*   Stack Frame        : Full (Frame Pointer in AR3)                         *
;*   Total Frame Size   : 2 Call + 2 Parm + 1 Auto + 0 SOE = 5 words          *
;******************************************************************************
_func:
      push      fp                   ; step 6    save old FP in local frame
      ldiu      sp,fp                ;           initialize new FP
      addi      1,sp                 ; step 7    allocate local frame for _x
      ldiu      2,r0
      ldiu      *-fp(3),r1           ;           load _d argument
      addi      *-fp(2),r1           ;           load _c argument and add it to _d
      sti       r1,*+fp(1)           ;           store the result into _x location
      addi      *+fp(1),r0           ;           compute _x
      sti       r0,*+fp(1)           ;           write it back (might be optimized)
                                     ;           r0 also contains the return value of the function
      ldiu      *-fp(1),r1           ;           load the return address into R1
      bud       r1                   ;           branch delayed to the return address
      nop
      subi      3,sp                 ; step 8    the local frame location, old FP and return address are
                                     ;           removed from the stack
      ldiu       *fp,fp              ;           restore the old FP into FP
                                     ; Branch Occurs to r1
```

Within this code are basically three sections: .cinit, .bss, and .text. The .cinit section contains tables with the values to initialize global variables. The .text section contains the program. In the C code are two global variables called c and d. In assembly the name is preceded with an underscore: _c and _d. In the .cinit section, these variables are defined using the specific format seen previously.

Example of a .cinit table definition:

```
.sect     ".cinit"                ; definitions of .cinit tables
.field    1,32                    ; there is one value (32 just indicates the length of the bit-field)
```

```
.field    _c+0,32              ; the value has to be copied to the location _c
.field    3,32                 ; the value which should be copied is 3
```

In the .bss section, the C compiler reserves memory space for the variables, which will be initialized either by the C boot routine or by a specific loader.

Example of a .bss definition:

```
.bss      _c,1           ; _c is the address of an un-initialized location
                         ; before entering the main() it will be initialized with the value 3 coming from
                         ; the .cinit section by a loader or the C boot routine
```

FP has been defined as a syntactic replacement of the AR3 register and is used as a reference pointer to address either arguments or automatic variables, i.e. variables that come into existence only when the appropriate function is called and disappear when the function is exited.

The C boot routine is called “boot.asm”, which is standard but can be replaced by a user-defined equivalent. ”boot.asm” should generally perform the following tasks:

❒ Reset the status register to zero.

❒ Initialize the stack pointer.

❒ Get the page and address of the .cinit section.

❒ Check whether the user selected the RAM or the ROM model.

❒ If the user selected the ROM model, then .bss will be auto-initialized; otherwise, this is done by an external loader.

❒ Set up the data page pointer (DP) to point to the .bss section (used only for small memory model).

❒ Call the C code (*call _main*).

This ensures that the entry point of the user C code is *_main* and is reached with a function call. Because it has been accessed with a call, the processor automatically saves on stack the return address of the caller (step 1). Now the called function has to save the previous Frame Pointer FP for the following reasons:

❒ To restore the context on exit (step 2)

❒ To initialize the FP to point to the old frame pointer (*ldiu sp, fp*)

❒ To allocate space in the local frame for the automatic variables a and b and initialize them

Because the FP actually points to the old FP, the first local variable *a* is addressed as *+fp(1) and the second variable *b* as *+fp(2) (*sti r1, *+fp(1)* or *sti r0, *+fp(2)* as it can be seen at step 3). Afterward, the global variable d is incremented and pushed together with c onto the stack as arguments (step 4). It is important to notice that arguments are pushed in reverse order; i.e., the rightmost declared argument is pushed first, and the leftmost is pushed last. This makes it straightforward to indirectly access the arguments through the FP because the leftmost has only an offset of minus two and the rightmost has the maximum offset. This method is called *standard runtime model*.

The call to func then occurs (step 5) and the return address is automatically saved onto the stack – this is the final action done by the caller. Now it is the responsibility of the called function to save the old frame pointer onto the stack (step 6) and set up the new Frame Pointer to the current sp (*ldiu sp, fp*). The local frame is allocated here by adding 1 to sp because there is only one local variable (step 7).

The called function can now access its arguments, which are addressed as *-fp(2) and *-fp(3) for _c and _d, respectively, and the _x value is computed and stored in the local frame (*sti r0, *+fp(1)*). The called function returns an integer and therefore the return value has to be located in r0. For more information, see the section, *The Return of a Function*.

The called function has executed its code and returns to the caller. This action is performed within four steps:

1) The return address is loaded into a register (*ldiu *-fp(1), r1*), the local frame and return address are de-allocated from stack (step 8), and the old FP is restored (*ldiu *fp, fp).* (A delayed branch to the return address r1 has already been performed.)

2) The program control is then passed back to the caller function.

3) The returned value of _func is stored in the local frame (*sti r0,*+fp(1)*).

   The local variable _b is computed according to the C code before being stored in the local frame (*sti r1, *+fp(2)*).

4) The two arguments that were passed through the stack (step 9) are de-allocated.

   The core loop (*for (;;)*) has been executed and the program branches delayed to the next iteration (*bud L2*), which will act the same way. In this example the loop is infinite and therefore the main routine will never exit.

The following table shows that the stack evolution has been recapitulated with the filling up of the stack on the left side and the release of data on the right side:

Stack (Top)	**Stack Use for Call**	**Step for Call**	**Step for Return**	**Stack Use for Return**
Free				
Free				
Free				
Free				
_x	Automatic variable _x is allocated (local frame)	7	8	Ffunction _func is returned and therefore the local frame (_x) is released
Old FP save	Save on entry has been done by the called function	6	8	Function _func is returned and therefore the old FP is released
Return address to the caller	Function _func is called and therefore the return address is automatically saved on stack by CALL	5	8	Function _func is returned and therefore the return address to the caller is released
_c	Argument _c is passed by stack	4	9	Argument _c that was passed is popped from stack
_d	Argument _d is passed by stack	4	9	Argument _d that was passed is popped from stack
2	Automatic variable _b is allocated (local frame)	3		
1	Automatic variable _a is allocated (local frame)	3		
Old FP save	Save on entry has been done by the called function	2		
Return address to C boot routine	Automatically save on stack by the CALL _main instruction of the C boot routine	1		
	Last occupied stack location before entering C	0		Came back to the old context
Stack (bottom)				

	This is the caller function responsibility
	This is the called function responsibility

The Register Argument Runtime Model

In the register argument runtime model, the C compiler uses dedicated registers to pass arguments to the called function. If the arguments do not fit within the dedicated register set because of type incompatibility or the number of registers has been exhausted, the remaining arguments are pushed onto the stack following the stack argument runtime model convention discussed in the previous section.

The six registers used to pass arguments are: ar2, r2, r3, rc, rs, and re. In a first pass, r2 and r3 are allocated to any floating-point numbers in the argument list starting from the left. In a second pass, the integer or pointer type arguments are allocated to the remaining registers in a left-to-right manner. An exception to these rules is the ellipsis declaration, such as in example function f3 in the table below, where the number of arguments is not fixed in the declaration and therefore the number of passed arguments must equal or exceed the number of function parameters. Here the last explicitly declared argument is passed on the stack so that its stack address acts as a reference for accessing the possible other arguments.

The examples below describe the allocation of parameters for function calls:

allocation type	**function definition**	**pass**
	int f0(int *a, int b, int c, int d, int e, int f, int g, int h);	
float arguments		1st pass left to right
ptr or integer arguments	ar2 r2 r3 rc rs re	2nd pass left to right
remaining arguments	Stack Stack	3rd pass right to left
global allocation	ar2 r2 r3 rc rs re Stack Stack	final
	int f1(int a, float b, int *c, struct A d, float e, int f, int g);	
float arguments	r2 r3	1st pass left to right
ptr or integer arguments	ar2 rc rs re	2nd pass left to right
remaining arguments	Stack	3rd pass right to left
global allocation	ar2 r2 rc rs r3 re Stack	final
	int f2(float a, int *b, float c, int d, float e);	
float arguments	r2 r3	1st pass left to right
ptr or integer arguments	ar2 rc	2nd pass left to right
remaining arguments	Stack	3rd pass right to left
global allocation	r2 ar2 r3 rc Stack	final
	int f3(struct x y, int b, int c, int d,...);	
float arguments		1st pass left to right
ptr or integer arguments	ar2 r2 r3	2nd pass left to right
remaining arguments	Stack Stack ...	3rd pass right to left
global allocation	ar2 r2 r3 Stack Stack ...	final

The following example uses C and assembly interface. The main program calls a function that calculates the sum of the figures from 0 to 10. This function is coded in assembly language and assumes that the register argument runtime model is in use to access arguments. The C program is named example1.c and the assembly program is called sum.asm. The command line for the C compiler must be:

```
cl30 -k  -mr example1.c sum.asm -z .....
```

The option -mr introduces the usage of the register argument runtime model. In the C code below, the s function is defined as external. From that the reference will be solved at link time. The exit() function is used to return to the caller of the main function, which is the boot routine.

Example1.c program:

```
extern int s(int a0,int a1,int a2,int a3,int a4,int a5,int a6,int a7,int a8,int a9);
        /* description of s function coded in C language int                          */
        /* s(int a0,int a1,int a2,int a3,int a4,int a5,int a6,int a7,int a8,int a9);   */
        /* int s(int a0,int a1,int a2,int a3,int a4,int a5,int a6,int a7,int a8,int a9) */
        /* { return(a0+a1+a2+a3+a4+a5+a6+a7+a8+a9); }                                  */
main()
{
        int c0=0,c1=1,c2=2,c3=3,c4=4,c5=5,c6=6,c7=7,c8=8,c9=9, sum;
        sum = s(c0,c1,c2,c3,c4,c5,c6,c7,c8,c9);
        exit(0);
}
```

The assembly-coded version of s is described below. The FP is defined as ar3 for coherency with C convention. The function s is referenced through _s because the compiler appends an underscore to the beginning of all identifiers. The global definition of _s defines the symbol as external and allows the linker to resolve references to it.

The first six arguments are assigned to the six registers used to pass arguments (ar2, r2, r3, rc, rs, re). The remaining arguments are pushed in reverse order onto the stack following the stack argument runtime convention. These actions are executed by the caller. The called function has only to preserve the old frame pointer and initialize the new one to access arguments that have been stacked. After the computation has been done, the function must restore the environment and return the result to the caller.

sum.asm example:

```
FP      .set    AR3
        .globl  _s

;>>>>   int s(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
******************************************************
* FUNCTION DEF : _s
******************************************************
_s:
        PUSH  FP
        LDI   SP,FP
*
* AR2   assigned to parameter a0
* R2    assigned to parameter a1
* R3    assigned to parameter a2
* RC    assigned to parameter a3
* RS    assigned to parameter a4
* RE    assigned to parameter a5
*
;>>>>   {return(a0+a1+a2+a3+a4+a5+a6+a7+a8+a9);
  ADDI    R2,AR2,R0       ;R0=a0+a1
  ADDI    R3,R0           ;R0+=a2
  ADDI    RC,R0           ;R0+=a3
  ADDI    RS,R0           ;R0+=a4
  ADDI    *-FP(2),R0      ;R0+=a6
  ADDI    *-FP(3),R0      ;R0+=a7
  ADDI    *-FP(4),R0      ;R0+=a8
  ADDI    *-FP(5),R0      ;R0+=a9
  LDI     *-FP(1),R1
  BD      R1         ;branch delay to the return address
  LDI     *FP,FP   ; take benefit of the branch delay to restore old FP
```

```
    ADDI    RE,R0     ; take benefit of the branch delay to perform the last sum and store the returned
                      ; result into R0 R0+=a5
    SUBI    2,SP      ; de-allocates old FP location and return address
```

The caller sequence generated by the compiler will look as follows:

```
          ........
          local frame management for
          local variables
          ........
;>>>>     sum=s(c0,c1,c2,c3,c4,c5,c6,c7,c8,c9);

          ldiu     ir0,r0                ; load c9 from temporary register to r0
          push     r0                    ; push r0 - this sequence might be optimized with -o
          ldiu     *+fp(9),r0            ; load c8 into r0
          push     r0                    ; push r0
          ldiu     *+fp(8),r0
          push     r0
          ldiu     *+fp(7),r0
          pus       r0
          ldiu     *+fp(1),ar2           ; pass the first six arguments through registers following the
                                         ; rules discussed in this paragraph
          ldiu     *+fp(6),re            ; load the remaining parameters to the appropriate registers
          ldiu     *+fp(4),rc
          ldiu     *+fp(5),rs
          ldiu     *+fp(2),r2
          ldiu     *+fp(3),r3
          call     _s
                         ; Call Occurs
```

If a function is used as an argument of a function, such as a quick-sort algorithm:

```
void qsort(void *v[], int left, int right, int (*comp)(void *, void *));
```

the address of the function will be passed accordingly as just another pointer type of argument.

Naturally the calling and the called function must be compiled with the same argument runtime model as well as with the same memory model. In an assembly code, this can be reflected using conditional programming with the predefined symbols, .REGPARM (==1 indicates register argument runtime model) and BIGMODEL (==1 indicates usage of big memory model).

The Return of a Function

If the return value of a function is a scalar type of data like an integer value, pointer, or floating-point number, it will be placed in the register r0. This is demonstrated in the sample file example0.c previously. If the register-argument runtime model is used, a pointer will be returned via the ar0 register. If the return value is a structure, memory space is allocated on the heap and the structure is copied to this memory space. The address of this structure is then passed via the ar2 register.

References

B.W. Kernighan, D.M. Ritchie: *The C Programming Language*; Prentice Hall 1988

TMS320C3x User's Guide; Texas Instruments 1997

TMS320C3x/C4x Optimizing C Compiler User's Guide; Texas Instruments 1997

INTERNET

www.ti.com

Register with TI&ME to build custom information pages and receive new product updates automatically via email.

TI Semiconductor Home Page
http://www.ti.com/sc

TI Distributors
http://www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

US TMS320

Hotline	(281) 274-2320
Fax	(281) 274-2324
BBS	(281) 274-2323
email	dsph@ti.com

Americas

Phone	+1(972) 644-5580
Fax	+1(972) 480-7800
Email	sc-infomaster@ti.com

Europe, Middle East, and Africa

Phone	
Deutsch	+49-(0) 8161 80 3311
English	+44-(0) 1604 66 3399
Francais	+33-(0) 1-30 70 11 64
Italiano	+33-(0) 1-30 70 11 67
Fax	+33-(0) 1-30-70 10 32
Email	epic@ti.com

Japan

Phone	
International	+81-3-3457-0972
Domestic	+0120-81-0026
Fax	
International	+81-3-3457-1259
Domestic	+0120-81-0036
Email	pic-japan@ti.com

Asia

Phone	
International	+886-2-3786800
Domestic	
Australia	1-800-881-011

Asia (continued)

TI Number	-800-800-1450
China	10811
TI Number	-800-800-1450
Hong Kong	800-96-1111
TI Number	-800-800-1450
India	000-117
TI Number	-800-800-1450
Indonesia	001-801-10
TI Number	-800-800-1450
Korea	080-551-2804
Malaysia	1-800-800-011
TI Number	-800-800-1450
New Zealand	+000-911
TI Number	-800-800-1450
Philippines	105-11
TI Number	-800-800-1450
Singapore	800-0111-111
TI Number	-800-800-1450
Taiwan	080-006800
Thailand	0019-991-1111
TI Number	-800-800-1450

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current and complete. TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements. Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office. In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1998, Texas Instruments Incorporated

TI is a trademark of Texas Instruments Incorporated.
Other brands and names are the property of their respective owners.