Assembly Language Interface 4

4.1 INTRODUCTION

This chapter describes how to interface assembly language code with C code.

You must be familiar with ADSP-21000 family assembly language and the C runtime model to make best use of this chapter. See the *ADSP-2106x SHARC User's Manual*, *ADSP-21020 User's Manual*, and the previous chapter of this manual.

There are several ways to incorporate assembly language statements into a program that contains code written in C:

- Write assembly language functions that are C-callable
- Use assembly language statements within a C function (by using the `asm()` in-line assembly construct)
- Associate C variables with assembly-language symbols

Throughout this chapter there are references to `asm_sprt.h`, a header file containing macros to help you write assembly language implementations of C functions. These macros are summarized at the end of this chapter. This header file is found in `$ADI_DSP\21K\INCLUDE`. If you use these macros, any file including the `asm_sprt.h` header file *must* be processed by the G21K compiler, and not by the ASM21K assembler only.

4.2 ASSEMBLY LANGUAGE FUNCTIONS

Assembly language functions may be linked with C functions. They are normally placed in files with the `.s` or `.asm` suffix. There are several issues involved in writing a C-callable assembly language function:

Using registers
Retrieving parameters
Calling other functions
Function entry
Function exit
Naming conventions for assembly-language symbols

4.2.1 Using Registers

There are some points to consider when using ADSP-21xxx registers in assembly language programming under the C runtime environment:

- Most registers of the processor are available to the assembly language programmer.

- Some registers have special purposes and must be used only for those purposes. For example, some registers need to contain fixed values for the compiler.

- If registers are used in assembly language, rules must be followed when the contents of registers are saved and restored.

- We recommend using the macros in `asm_sprt.h` for saving registers to the stack and restoring them from it.

4.2.1.1 Special Purpose Registers

Two registers, called the *stack pointer* and *frame pointer,* are used to manipulate the C runtime stack. The C runtime stack is used to store automatic variables, pass parameters, store function return address, and store the intermediate results of computations.

The stack pointer, register I7, points to the *top of the stack.* The top of the stack is the next empty location on the stack. The stack grows towards address 0. Thus, if a value is “pushed” on the stack, it is placed at the location pointed to by the stack pointer and the stack pointer is decremented.

The frame pointer, register I6, points to the start of the frame for the current function.

4.2.1.2 Fixed Value Registers

M Registers

Several M registers are used to hold fixed values. The run-time header, which executes on processor startup, sets these registers to their designated values and calls `main()`. The compiler assumes that these registers contain their designated values. Assembly language functions must not change these registers, but may rely on their containing the following values:

DAG1 Register	DAG2 Register	Value
M5	M13	0
M6	M14	1
M7	M15	-1

L Registers

The compiler requires that the L registers contain zeros. All the L registers, **except** for L6 and L7, may be set by an assembly language function to any value. However,they must be reset to 0 before the assembly language function returns.

4.2.1.3 Saving & Restoring Registers

The compiler makes assumptions about how functions treat registers. If the compiler knows that a function does not destroy the contents of a register, the compiler may keep “live” data in that register when the function call is made. However, if the compiler expects that a subroutine destroys the contents of a register, the compiler attempts to avoid keeping useful information in that register when a function call is made. If the register contains “live” data, then the compiler must save a copy of that data before the function call and restore that copy to the register after the function call.

There are two classes of registers:

- Compiler *registers are* registers that the compiler assumes are preserved across function calls.

- *Scratch registers* are registers that the compiler assumes are not preserved across function calls.

  Note: It is not necessary that the called function actually overwrite *scratch* registers; to be safe, the compiler assumes that they are overwritten.

Registers	Compiler Registers	Scratch
Data Registers	R3, R5, R6, R7, R9, R10, R11, R13, R14, R15	R0, R1, R2, R4, R8, R12
Index Registers	I0, I1, I2, I3, I5, I8, I9, I10, I11, I14, I15	I4, I12
Modify Registers	M0, M1, M2, M3, M8, M9, M10, M11	M4, M12
Other Registers	MRF, MRB, MODE1, MODE2, USTAT1, USTAT2	

Here are some rules about saving and restoring registers:

1. Registers may be saved by pushing them onto the stack. This is done as follows:

```
dm(i7,m7)=r3;
```

   This instruction places register `R3` onto the stack and decrements the stack pointer.

   Note: Register M7 is fixed at -1.

2. A value may be popped off the stack by reading it, and then adjusting the stack pointer. The following instructions might be used:

```
r3=dm(1,i7);
modify(i7,1);
```

3. The stack pointer must always point to the next empty location on the stack.

   Note: The one exception to this rule is during the delay branch slots of the jump at the end of a function. The hardware of the ADSP-210xx locks out interrupts during these cycles, so you don't need to worry about the stack being corrupted.

As a negative example, the following code is not recommended because an interrupt could corrupt the stack.

```
modify(i7,1); /* NOT recommended */
r3=dm(0,i7);  /* NOT recommended */
```

If an interrupt occurred after the modify but before the read the stack would be pointing at valid data. The interrupt service routine would write over this data.

4. At the beginning of an assembly language function, all compiler registers that are used in the function must be saved. At the end of an assembly language function, all those that were saved must be restored.

5. Before a function call, all scratch registers that contain “live” data must be saved. After a function call, all those scratch registers that were saved must be restored.

4.2.1.4 Macros For Stack Access

The header file `asm_sprt.h` includes many macros useful for interfacing C to assembly language. For example, the `puts()` macro does a push:

```
puts=r7;
```

This instruction pushes `r7` onto the stack. Similarly the `gets()` macro reads off the stack. For example, to read the most recently pushed value into register `r9`, use this code:

```
r9=gets(1);
```

You can use the `restore_reg` macro in conjunction with the `save_reg` macro to save and restore all register file registers (R0-R15). The macros are found in `asm_sprt.h` . Use them as a templates for constructing code for performing register saves and restores.

4.2.1.5 Secondary Register Set

The register file of ADSP-21000 family processors has a complete secondary set of the primary registers. The C runtime environment model does not use any of the secondary registers. Therefore, you may use the secondary register set in assembly language freely. The C runtime environment is not corrupted by using the secondary register set. When you switch back to using the primary register set, the primary registers are as you left them.

4.2.2 Retrieving Parameters

This section describes how parameters to C functions are accessed.

4.2.2.1 Where The Parameters Are

In the C environment, arguments are passed to functions by placing them in registers or on the stack, according to the following rules:

1. Up to three arguments may be passed in registers.

   The first argument to be passed in a register is placed in `R4`.

   The second argument to be passed in a register is placed in `R8`.

   The third argument to be passed in a register is placed in `R12`.

2. Once one argument has been passed on the stack, all remaining arguments (those to the right) are on the stack.

3. All values wider than 32 bits are passed on the stack. These include variables of type `double` and `complex`, and structures passed by value.

   Whenever a `double` or `float` is placed on the stack, the most significant word falls at the lower address, the least significant word at the higher address.

   Whenever a `complex` is placed on the stack, the real part is put at the lower address, the imaginary part is put at the higher address.

4. The last named argument in a function call with a variable number of arguments is passed on the stack.

4.2.2.2 Parameter Passing Examples

Consider the following function prototype example:

```
 foo(int a, float b, char c, float d);
```

The first three arguments, `a`, `b`, and `c` are passed in registers `R4`, `R8`, and `R12`, respectively. The fourth argument `d` is passed on the stack.

This next example illustrates the effects of passing doubles.

```
 bar(int a, double b, char c, float d);
```

The first argument `a` is passed in `R4`. Since the second argument `b` is a multi-word argument, it is passed on the stack. As a result, the remaining arguments, `c` and `d`, are passed on the stack.

The following illustrates the effects of variable arguments on parameter passing:

```
 test(float a, int b, char c,...);
```

Here, the first two arguments, `a` and `b`, are passed in registers `R4` and `R8`. Since `c` is the last named argument, it is passed on the stack, as are all remaining variable arguments.

4.2.2.3 Accessing Stack Parameters

When arguments are placed on the stack, they are pushed on from right to left. The right-most argument is at a higher address than the left-most argument passed on the stack.

The following example shows how to access parameters passed on the stack:

```
test( int a, char b, float c, int d, int e, long f);
```

Parameters a , b , and c are passed in registers because they are single-word parameters. The remaining parameters, d, e, and f , are passed on the stack.

All parameters passed on the stack are accessed relative to the stack pointer, register I6 . The first parameter passed on the stack, d , is at address sp + 1 . To access it, you could use this assembly language statement,

```
r3=dm(1,i6);
```

The second parameter passed on the stack, e , is at sp + 2 and can be accessed by the statement

```
 r3=dm(2,i6);
```

The third parameter passed on the stack, f , is a long that has its most significant word at sp + 3 and its least significant word at the top of the stack. f(MSW) can be accessed by the statement

```
 r3=dm(3,i6);
```

4.2.2.4 Macros For Parameters

The `asm_sprt.h` file includes a macro, `reads()`, for reading parameters off the stack. For example, to read the second stack-passed parameter into register R5, you could use this statement:

```
 r5=reads(2);.
```

4.2.3 Calling Functions

You must follow the calling protocol to call a function in the C environment. The macros in `asm_sprt.h` are provided to make this easier.

4.2.3.1 The Calling Protocol

Calling a function in a C environment involves several steps:

1. The arguments to the function must be passed.

2. The return address must be stored on the stack. The return address must be the address **immediately preceding** the address where execution is to resume.

3. A delayed branch jump to the function call must be made. A jump is used, instead of a call, because the return is handled by another, indirect jump. Jumps are preferable to calls because calls are limited by the on-chip PC stack depth. Jumps have no nesting limit, as they allow saving the return address in external memory. The delayed branch form of the jump instruction is used so that the frame pointer adjustment may take place in the two delayed branch slots, which cannot be interrupted.

4. The frame pointer must be adjusted. The current function's frame pointer, `I6`, is written into `R2`, and then the the current stack pointer, `I7`, is written into `I6` to create the called function's frame pointer.

5. When the function returns, it may be necessary to adjust the stack pointer to remove the function's arguments from the stack. This is done by adding a constant, the number of stack positions used by arguments, to the stack pointer.

The calling protocol is different for calling C functions from assembly language routines. This does **not** affect C-callable assembly routines. The calling sequences for the ADSP-21020 and ADSP-2106x are different.

Use the following code sequence to call a function (foo()) for the **ADSP-21020**:

```
R2 = I6;               /* Hold old frame        */
I6 = I7;               /* Swap stack and frame  */
JUMP (PC, _foo) (DB);  /* JUMP to foo()         */
DM(I7, M7) = R2;       /* Save old frame        */
DM(I7, M7) = PC;       /* Save return address   */
```

Use the following code sequence to call a function (foo()) for the **ADSP-2106x**:

```
CJUMP _foo (DB);       /* JUMP to foo(), swap   */
DM(I7, M7) = R2;       /* Save old frame        */
DM(I7, M7) = PC;       /* Save return address   */
```

The old frame and return address are saved by the caller, so the called prologue is composed only of register saves.

The epilogue is:

```
I12 = DM (-1, I6);     /* Fetch return address  */
```

followed by any register restore operations. The compiler reads the return address before restoring other registers.

The last instructions of an **ADSP-21020** routine are:

```
JUMP (M14, I12) (DB);  /* Return to caller      */
I7 = I6;               /* Clean stack           */
I6 = DM (0, I16);      /* Restore old frame     */
```

The last instructions of an **ADSP-2106x** routine are:

```
   JUMP (M14, I12) (DB);  /* Return to caller      */
   RFRAME;                /* Restore stack, frame  */
   NOP;                   /* Used for useful op!   */
```

The compiler will replace the NOP with a useful instruction (such as a register restore operation) whenever possible.

4.2.3.2 Macros For Calling A Function

The `asm_sprt.h` file includes macros to perform all the necessary steps for calling C functions.

To push a value on the stack, use the `puts()` macro. For example to push the value of register `R3` onto the stack, you would use this statement:

```
puts=R3;
```

To call a function use the `ccall()` macro. For example, to call `foo()`,

```
ccall(_foo);
```

The `ccall()` macro pushes the return address onto the stack, performs the frame pointer adjustment, and jumps to the other function.

The `alter()` macro can be used to remove values from the stack. For example, to remove the last three values from the stack, you could use this statement:

```
 alter(3);
```

4.2.4 Function Entry

You must follow the C runtime environment calling protocol when entering a function. The macros in `asm_sprt.h` are provided to make this easier.

4.2.4.1 What Is Needed On Function Entry

On function entry, the called function must save information necessary to return to the calling context.

1. The calling function's frame pointer was loaded into register `R2` by the calling function (see the previous section). The old frame pointer must be saved to the stack so that it can be used later.

2. The calling function pushes the return address onto the stack. This value must be saved so that it can be used later.

4.2.4.2 Macros For Entry

The `entry` macro, found in the header file `asm_sprt.h`, saves both the calling function's frame pointer and the return address.

4.2.5 Function Exit

4.2.5.1 What Is Needed On Function Exit

Several operations must be performed at the end of an assembly language function.

1. The return value must be placed in the appropriate register(s). If a single word value is being returned, it must be returned in register R0. If a two word value is being returned, it must be returned in registers R0 and R1.

   If a `double` is returned, `R0` contains the MSW (Most Significant Word) and `R1` contains the LSW (Least Significant Word).

   If a `complex` is returned, `R0` contains the real part and `R1` contains the imaginary part.

2. The values that occupied compiler registers that were saved at the top of the function must be restored to their original locations. They may be read off the stack, relative to the frame pointer.

3. The calling function's stack pointer must be restored. Before transferring control, the calling function copied its stack pointer, `I7` , to the frame pointer, `I6`. To restore the calling function's stack pointer, copy the frame pointer, `I6`, to the stack pointer, `I7`.

4. The calling function’s frame pointer must be restored. Previously, it was transferred by the calling function from `I6` to `R2`. At the top of the called function, it was then moved to the stack. It must now be restored to `I6`.

5. Control must be returned to the calling function. The return address, which was saved by the called function onto the stack, must be restored to a DAG2 I register. Then, an indirect jump may be made to this register plus one.

4.2.5.2 Macros For Return

Steps 3 – 5 described above are incorporated into a single macro, exit. The exit macro reads the return address off the stack, performs the stack and frame pointer adjustments, and returns control to the calling function.

4.2.6 Leaf Functions

The definition of a Leaf function is a function that never calls other functions. There are some optimizations that can be performed with leaf functions that are not permissible with non-leaf functions. Specifically, in a leaf function it may not be necessary to save either the calling function’s frame pointer or the return address onto the stack. The macros `leaf_entry` and `leaf_exit` are analogous to the macros `entry` and `exit`, but are more efficient.

Warning: These macros do not save or restore the register `R2`— do not destroy the contents of `R2` if these macros are used.

4.2.7 Naming Conventions For Assembly Language Symbols

In order for C functions to link with assembly functions, use the `.global` and `.extern` assembly language directives. These are fully described in the *ADSP-21000 Family Assembler Tools Manual*. C language names and variables are prefixed with an underscore when used in assembly language.

The following example shows the use of C and assembly functions linked together.

C code:

```
void asm_func(void);           /* assembly and c functions */
                               /* prototyped here */
void c_func(void);

int c_var=10;                  /* c_var defined here as a */
                               /* global; used in .asm file */
                               /* as _c_var */

extern int asm_var;            /* asm_var defined in .asm */
                               /* file as _asm_var */
main () {
  asm_func();                  /* call to assembly function */
  }

void c_func(void) {            /* this function gets called */
                               /* from asm file */
  if (c_var != asm_var)
    exit(1);
  else
    exit(0);
}
```

Assembly code:

```
#include <asm_sprt.h>
.segment/dm seg_dmda;
.var _asm_var=0;          /* asm_var is defined here */
.global _asm_var;         /* needed so C function can see it
*/
.endseg;

.segment/pm seg_pmco;
.global _asm_func;        /* _asm_func is defined here */
.extern _c_func;          /* c_func is defined in C file */
.extern _c_var;           /* c_var is defined in C file */

_asm_func:
  entry;                  /* always use entry macro from
                              asm_sprt.h first */

  r8=dm(_c_var);          /* access the global C variable */
  dm(_asm_var)=r8;        /* set asm_var to c_var */

  ccall(_c_func);         /* make a call to the C function */

  exit;                   /* exit macro from asm_sprt.h */

.endseg;
```

Listing 4.1 C & Assembly Linked Together

4.2.8 Examples

This section reiterates by example the concepts discussed so far in this chapter.

4.2.8.1 Simple Assembly Routines

The simplest set of assembly routines are those with no arguments and no return values. An assembly routine like this might wait for an external event, or delay a number of cycles available in a global variable. In such assembly routines pay close attention to register usage. The assembly routine must save and later restore any compiler registers that are modified. Since a simple delay does not need many registers, you can use scratch registers which do not need to be saved.

```
/* void delay ( void );
   An assembly language subroutine to delay N cycles
   where N is the value of the global variable del_cycle */

 #include <asm_sprt.h>;

.segment/pm seg_pmco;
.extern _del_cycle;
.global _delay;
_delay:
   leaf_entry;    /* this must appear as the first line */
                  /* in every assembly language routine */

   R4 = DM ( _del_cycle);
                  /* we use r4 because it is a scratch */
                  /* register and doesn't need to be */
                  /* preserved */

   LCNTR = R4, DO d_loop UNTIL LCE;
d_loop:     nop;

   leaf_exit;     /* The exit macro is the last line */
                  /* executed in any assembly language */
                  /* subroutine. The exit macro returns */
                  /* control to the calling program */
 .endseg;
```

Listing 4.2 Delay N Cycles

Note that all symbols accessed from C contain a leading underscore. Since the assembly routine name, `delay`, and the global variable `_del_cycle` are both available to C programs, they contain a leading underscore in the assembly code listing.

4.2.8.2 Assembly Routines With Parameters

Another, more complicated set of routines are those with parameters but no return values. The following example adds five inputs integers passed as parameters to the function.

```
/* void add5 (int a, int b, int c, int d, int e);
 An assembly language subroutine that adds 5 numbers */

#include <asm_sprt.h>

.segment/pm seg_pmco;
.extern _sum_of_5;      /* variable where sum will be stored */

.global _add5;

_add5:
leaf_entry;

/* the first three parameters are passed in r4, r8, r12, respectively */
r4=r4+r8;               /* add the first and second parameter */
r4=r4+r12;              /* add the third parameter */

/* the fourth/fifth parameters can be accessed by reads(1)/reads(2) */
r8=reads(1);            /* put the fourth parameter in r8 */
r4=r4+r8;               /* add the fourth parameter */
r8=reads(2);            /* put the fifth parameter in r8 */
r4=r4+r8;               /* add the fifth parameter */

dm(_sum_of_5)=r4;       /* place the answer in the global variable */

leaf_exit;
.endseg;
```

Listing 4.3 Add5

4.2.8.3 Assembly Routines With Return Values

Another class of assembly routines are those which have both parameters and return values. A simple example of such an assembly routine would be to add two numbers and return their sum. Return values are stored in the `R0` register.

```
/* int add2 (int a, int b);
   An assembly language subroutine that adds two numbers and
   returns sum */

#include <asm_sprt.h>

.segment /pm seg_pmco;
.global _add2;
_add2:
leaf_entry;

/* the first two parameters passed in r4, r8, respectively */
/* return values are always returned the r0 register */

r0=r4+r8; /* add the first and second parameter, store in r0*/

leaf_exit;
.endseg;
```

Listing 4.4 Add Two Integers

4.2.8.4 Non-Leaf Assembly Routines

A more complicated example, one which calls another routine, would be to compute the root mean square of two floating point numbers ($z=(x^2+y^2)^{1/2}$). While it is simple to calculate a square-root in ADSP-21000 assembly language, this example uses the square root function provided in the C run-time library. It illustrates how to call C functions from assembly language.

```
/* float rms ( float x, float y) ;
An assembly language subroutine to return the rms
z = (x^2 + y^2)^(1/2)  */

#include <asm_sprt.h>

.segment /pm seg_pmco;
.extern _sqrtf;
.global _rms;
_rms:

entry;

f4=f4*f4;
f8=f8*f8;
f4=f4+f8;

/* f4 contains argument to be passed to sqrtf function */

/* use the ccall macro to make a function call in a C environment */

ccall (_sqrtf);

/* f0 contains the result returned by the sqrtf function. We need to
return the result in f0, and it is already there */

exit;
.endseg;
```

Listing 4.5 Root Mean Square

If a called function takes more than three single word parameters, the remaining parameters must be pushed on the stack, and popped off the stack after the function call. The following function could call the `add5` routine (Listing 4.3) described previously.

```
/* int calladd5 ( void ) ;
An assembly language subroutine that calls another routine with
more than 3 parameters. Here we add the numbers 1,2,3,4,5. */

#include <asm_sprt.h>

.segment /pm seg_pmco;
.extern _add5;
.extern _sum_of_5;
.global _calladd5;
_calladd5:

entry;
r8=2;  /* the second parameter is sent in r8 */
r12=3; /* the third parameter is sent in r12 */
r4=4;  /* the fourth parameter is stored in r4 for pushing onto stack */
puts=r4; /* put fourth parameter in stack */
r4=5;    /* the fifth parameter is stored in r4 for pushing onto stack */
puts=r4; /* put fifth parameter in stack */
r4=1;    /* the first parameter is sent in r4 */

/* use the ccall macro to make a function call in a C environment */
ccall (_add5);
alter(2);            /* remove the two arguments from the stack */
r0=dm(_sum_of_5);    /* _sum_of_5 is where add5 stored its result */
exit;
.endseg;
```

Listing 4.6 Call Add5

Some functions need to make use of compiler registers. A variable must be stored in a compiler register whenever:

1. Its lifetime spans a function call, or

2. There are no more scratch registers available.

The following is an example of an assembly routine that performs an operation on the elements of a C array.

```
/* void foo ( float function(float), float *array, int length);
    An assembly language routine that operates on a C array */

#include <asm_sprt.h>

.segment/pm seg_pmco;
.global _foo;
_foo:
entry;
puts=i8; /* We use i8, a compiler register, since we don't
            want to have to store it for every function
            call. Compiler registers are guaranteed to be
            preserved across function calls */
r0=i1;
puts=r0; /* we also need to save i1, for the same reason */

i8=r4;   /* read the first argument, the address of the function to call */
i1=r8;   /* read the second argument, the C array containing the data to
            be processed */
r0=r12;  /* read third argument, the number of data points in the array */

lcntr=r0, do foo_loop until lce;   /* loop through data points */

f4=dm(i1,m5);     /* get data point from array, store in f4 for parameter
                     for function call */
ccall(m13,i8);           /* call the function */
foo_loop: dm(i1,m6)=f0;  /* store the return value back in the array */
i1=gets(1);              /* restore the value of i1 */
i8=gets(2);              /* restore the value of i8 */
exit;
.endseg;
```

Listing 4.7 Array Operation

4.2.8.5 A Comprehensive Example

Here is an example of a C-callable assembly language function. This function computes the dot product of two vectors. The two vectors and their lengths are passed as arguments. Since scratch registers are used for intermediate values and indirect addressing, no registers need to be saved or restored.

```
/* dot(int n, dm float *x, pm float *y); Computes dot product of two floating
point vectors of length n, one in dm one in pm. n must be greater than 2.*/

#include <asm_sprt.h>

.segment/pm seg_pmco;
                  /* The name of a function is formed by
                  taking its C name and prepending an underscore */

.global _dot;
_dot:

   entry;          /* Save old frame pointer and return address */


   r0=r4-1,i4=r8;  /* Load first vector address into I
                      register, and load r0 with length-1 */

   r0=r0-1,i12=r12; /* Load second vector address into
                       I register and load r0 with length-2
                      (because we're doing 2 iterations outside
                       by feeding and draining pipe */

   f12=f12-f12,f2=dm(i4,m6),f4=pm(i12,m14);
   /* Zero the register that will hold the result and start feeding pipe */

   f8=f2*f4, f2=dm(i4,m6),f4=pm(i12,m14);
   /* Second stage of pipeline, also do multiply */

   lcntr=r0, do dot_loop until lce;  /* Loop length-2 times, three-stage
                                        pipeline: read, mult, add       */

dot_loop:
   f8=f2*f4, f12=f8+f12,f2=dm(i4,m6),f4=pm(i12,m14);
   f8=f2*f4, f12=f8+f12;                              /* drain the pipe */

   f0=f8+f12;                          /* and end with the result in r0,
                                          where it'll be returned       */

 exit;      /* need to restore the old frame pointer and return control */
.endseg;
```

Listing 4.8 Dot Product

4.3 IN-LINE ASSEMBLY LANGUAGE WITH ASM()

Some assembly language statements cannot be expressed easily or efficiently with C constructs. For this reason, G21K allows a programmer to express instructions in assembly language within a C function using the `asm()` construct.

A simple example of the `asm()` construct:

```
asm(“bit set mode2 0x20;”);
```

In an assembler instruction using `asm()`, you can specify the operands of the instruction using C expressions. You do not need to know which registers or memory locations contain C variables. Used in this way, `asm()` takes the following form:

asm(*template*:*output operands*:*input operands*:*clobbered registers*);

For example, here is how to use the ADSP-210xx `clip` instruction:

```
{

 int x,y, result;
 ...
 asm (“%0=clip %1 by %2;” : “=d” (result) : “d” (x), “d”
(y));
}
```

Here `x` and `y` are the C variables for the input operands, while `result` is the C variable for the output operand. The letter `d` is the *operand constraint* for the variables. Each variable has `d` as its operand constraint, indicating that a data register, R0 - R15, is required. The “=” in “=`d`” indicates that the operand is an output. Operand constraints are further explained in Section 4.3.2.1.

Further details of `asm()` are explained below.

4.3.1 Template

Only the first argument to `asm()` is mandatory. The first argument to `asm()` is the *assembly language template*. The template is a string that describes the instruction that is inserted into the assembly language stream, including where the operands are placed.

In the above example, the template is “`%0=clip %1 by %2;`”. The `%0` is replaced with operand zero, the first operand. The `%1` and `%2` are replaced with operands one and two respectively.

- Everything except the template is optional.
- Each operand is described by an operand-constraint string followed by a C expression in parentheses.
- A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input operand, if any.
- A colon separates clobbered registers from the input operands.
- Commas separate operands within arguments.
- You cannot have more operands than you have in your template.
- The total number of operands is limited to ten.
- If there are no output operands, and there are input operands, then there must be two consecutive colons separating the assembly template from the input operands.
- Output operand expressions must be *lvalues*; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is a valid assembler input.

4.3.2 Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. There are several pieces of information that need to be conveyed. First, which registers are allowed for each operand. Second, whether that operand is an input to or an output from the instruction. Third, which variables are represented by the operands.

4.3.2.1 Constraint Letters

The operand constraint string contains a letter corresponding to each class of register that is allowed. The following table describes the correspondence between constraint letters and register classes.

`r`	any file register	(`Rx, Ix, Mx, MRF, MRB`)
`d`	`r0 - r15`	
`k`	`r0 - r3`	
`b`	`r4 - r7`	
`c`	`r8 - r11`	
`l`	`r12 - r15`	
`w`	`i0 - i7`	(DAG1 `I`-registers)
`x`	`m0 - m7`	(DAG1 `M`-registers)
`j`	`l0 - l7`	(DAG1 `L`-registers)
`h`	`b0 - b7`	(DAG1 `B`-registers)
`y`	`i8 - i15`	(DAG2 `I`-registers)
`z`	`m8 - m15`	(DAG2 `M`-registers)
`e`	`l8 - l15`	(DAG2 `L`-registers)
`a`	`b8 - b15`	(DAG2 `B`-registers)
`f`	`MRF MRB`	(Accumulators)
`u`	`USTAT1` `USTAT2`	(User registers)

Note: The use of any other letter not specified above will result in unspecified behavior.

Note: The compiler does not check the validity of the assembly code against the constraint letter specified.

Each operand in the example has a "`d`" for an operand constraint, signifying that any register R0 through R15 can be used.

4.3.2.2 Assigning Inputs &Outputs

To best assign registers to the operands, the compiler must be told which operands in an assembly language instruction are inputs and which are outputs. The compiler is told this in two ways, both required.

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and always follow the output operands.

- The operand constraints describe which registers are set because of an assembly language instruction. In the above example, the "=" in "=d" indicates that the operand is an output; all output operands must use =.

4.3.3 Clobbered Registers

Some instructions overwrite specific hardware registers. To describe this, put a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is an example:

```
int value = 10;
asm( “r9=%0;” : /* no outputs */ : “d” (value) : “ r9” );
```

If you refer to a particular hardware register from the assembler code, then you must list the register after the third colon to tell the compiler that the register's value is modified.

4.3.3.1 Number Of Instruction Per Template

There can be many assembly instructions in one template. The input operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. If the `asm()` string is longer than one line, you may continue it on the next line by placing a backslash (\) at the end of the line. Here is an example of multiple instructions in a template:

```
asm (“r9=%1; \
   r10=%2; \
   %0=r9+r10;”
   : “=d” (result)          /* output */
   : “d” (from), “d” (to)    /* input */
   : “r9”, “r10”);           /* clobbers */
```

4.3.3.2 The & Constraint Modifier

Unless an output operand has the & constraint modifier, G21K may allocate it in the same register as an unrelated input operand. This is because G21K assumes that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use & for each output operand that may not overlap an input.

To use the & constraint modifier, rewrite the example above using:

```
 : “=&d” (result)
```

4.3.4 Reordering & Optimization

If an asm() has output operands, G21K assumes, for optimization purposes, the instruction lacks side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful. The compiler may eliminate them if the output operands are not used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an asm() instruction from being deleted, moved significantly, or combined, by writing the keyword volatile after the asm. For example:

```
#define set_priority(x) \
asm volatile ("set_priority %0;": /* no outputs */ : "d" (x))
```

Note: Even a volatile asm() instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You cannot expect a sequence of volatile asm() instructions to remain perfectly consecutive. If you want consecutive output, use a single asm() construct. Another way to avoid reordering is to use the output of an asm() construct in a C instruction.

Even with these precautions, be aware that the -O and -O2 compiler switches may cause asm() instructions to be moved. Compile without optimization to minimize the chance of reordering.

4.3.5 asm() Statements Containing Macros

If you want to use macros inside asm() statements, you must specify an include file that defines the macros. The following example illustrates the method for including the def21060.h file and using a macro. The def21060.h file is in the ADI_DSP/21k/include directory and contains bit definitions for ADSP-2106x SHARC system registers:

```
asm("#include <def21060.h>");

main() {
 asm("bit set mode2 TIMEN;");
}
```

4.3.6 Assembler Instructions With Input/Output Operands

The output operands must be write-only; G21K assumes that the values in these operands before the instruction are “dead” and need not be generated.

When the assembler instruction has an operand that is both read from and written to, you must logically split its function into two separate operands: one input operand and one write-only output operand. The connection between them is expressed by constraints that say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the `modify` instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm (“modify (%0,%2);” : “=w” (foo) : “0” (foo), “x” (bar));
```

The constraint `“0”` for operand 1 says that it must occupy the same location as operand 0. A digit in an operand constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand is in the same place as another. Just because `foo` is the value of both operands does not guarantee that they are in the same place in the generated assembler code. The following does not work:

```
/* NOT recommended */
asm (“modify (%0,%2);” : “=w” (foo) : “w” (foo), “x” (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register. Since the register for operand 1 is not in the assembler code, the code does not work.

Be aware that asm() does not support input operands that are both read from and written to. The example below shows a **dangerous** use of such an operand. In this example, arr is modified during the asm() operation. The compiler only knows that the output, result_asm , has changed. Subsequent use of arr after the asm() instruction may yield incorrect results since those values may have been modified during the asm() instruction and not restored.

```
/* NOT recommended */
int result_asm;
int *arr;
asm(“%0=dm(%1,3);” : “=d” (result_asm) : “w” (arr));
```

4.3.7 Macros Containing asm()

Usually the most convenient way to use these asm() instructions is to encapsulate them in macros that look like functions. For example,

```
#define clip_macro(result,x,y) \
 asm(“%0=clip %1 by %2;” : “=d” (result) : “d” (x), “d” (y));

main () {
 int result_var;
 int x_var=10;
 int y_var=2;
 clip_macro(result_var, 10, 2);
 /* or */
 clip_macro(result_var, x_var, y_var);
}
```

This defines a macro, clip_macro(), which uses the asm() instruction to perform an assembly-language clip operation of variable x_var by y_var, putting the result in result_var.

4.4 ASSOCIATING C VARIABLES WITH SYMBOLS

It is possible with G21K to supply an assembly language description for the storage or naming of C variables. For example, you may wish to place variables into particular memory segments defined in the architecture file. Or, you may wish that a variable use a particular assembly language symbol or register.

4.4.1 Assembly Language Symbols

Normally, the assembly language symbol assigned to a global variable consists of an underscore (_), with the global variable's C-name appended. For example, the C global variable `foo` is placed at the address referred to by the assembly language `symbol _foo`.

It is possible to assign an assembly language symbol to a static or global variable using `asm()`. Wherever the variable is defined or declared, place an asm("name") after the definition or declaration. The assembly language symbol must be placed within double quotes, surrounded by parentheses, after the `asm` keyword. For example,

```
int foo asm("bar");
```

This specifies that the name used for the variable `foo` in the assembler code is `bar` instead of the usual `_foo`.

This feature lets you define names for the linker that do not start with an underscore. You must make sure that the assembler names you choose do not conflict with any other assembler symbols.

4.4.2 Assigning Registers

You can also use the `asm()` construct to assign a particular register to a variable. Place the register name within quotes as the argument to `asm()`. For example,

```
register int x asm("r9");
```

assigns the register `r9` to the variable `x`.

Note: Register names are case sensitive—use lower case: `r` not `R`.

Some points to consider:

- Global register variable declarations **must** precede function definitions.
- Choose a register that is normally saved and restored by function calls, so that library routines do not overwrite it.

- Defining a global register variable in a certain register reserves that register entirely for this use, within the current file. The register is not allocated for any other purpose in the functions in the current file. The register is not saved and restored by these functions.

- It is not safe to access the global register variables from signal handlers, or from more than one thread of control. The runtime library routines may temporarily use the register.

- It is not safe for one function that uses a global register variable to call another such function by way of a third function that was compiled without knowledge of this variable (that is, in a different source file in which the variable was not declared).

- Excessive use of this feature may leave the compiler too few available registers to compile certain functions.

- This feature may not be used with the `-g` compiler switch for debugging.

You can define a local register variable with a specified register like this:

```
register int *foo asm (“i5”);
```

Here `i5` is the name of the register that must be used. Note that this is the same syntax used for defining global register variables, but for a local variable it appears within a function.

Defining such a register variable does not reserve the register. It remains available for other uses in places where flow control determines the variable’s value is not live.

The `CIRCULAR_BUFFER()` macro in `macros.h` makes use of this feature. The C examples `circbuf.c` and `circbuf1.c` in Appendix A shows the use of this macro.

4.5 ASSEMBLY SUPPORT MACROS

This section lists the macros contained in `asm_sprt.h`. The files are found in `$ADI_DSP\21K\INCLUDE`. These macros are recommended for interfacing assembly-language routines with C functions, and can help you write assembly language implementations of C functions. You may wish to print a listing of the `asm_sprt.h` file to see how these macros were written.

If you use these macros, any file including the `asm_sprt.h` header file *must* be processed by the G21K compiler, and not by the ASM21K assembler only.

Note: Section 4.2.6 explains leaf assembly routines—assembly routines that do not call other assembly routines.

`entry`	The previous frame pointer is saved in R2 by the calling function. R2 is pushed onto the stack. The return address (minus one) is also pushed onto the stack. This macro must be the first line in any non-leaf assembly routine.
`exit`	The previous Frame Pointer is restored from the current Stack Pointer. The previous Stack Pointer is restored from the stack. The return address is restored from the stack and a JUMP is issued to return control to the calling assembly routine. Note that the restoration of the Stack and Frame Pointers is after the JUMP, but the JUMP is delayed (DB) so that the two instructions immediately following are executed. This macro must be the last line of any non-leaf assembly routine.
`leaf_entry`	Currently `leaf_entry` takes no action. The previous Stack Pointer and return address are in registers on assembly routine entry that are only corrupted on a subsequent assembly routine call. If you do not corrupt these registers, you do not need to save them. This macro must be included as the first line of a leaf assembly routine, though it is currently empty, so that future changes to the runtime model can be incorporated without modifying source code.

`leaf_exit`	See `exit`. Since no assembly routine was called, the registers are intact and do not need to be loaded from the stack. This macro must be the last line in a leaf assembly routine.
`ccall(x)`	To call a C language function the Frame Pointer must be saved in R2 and the Stack pointer set to the Frame Pointer. The return address (minus one) is stored on the stack and a JUMP is made to function `x`.
`reads(x)`	A value is read from the stack, offset from the Frame Pointer by `x` memory locations. The parent assembly routine can access passed parameters by substituting the parameter number (the first is 1) for `x`. `reads(x)` also may be used to restore pushed registers at the end of an assembly routine (using a negative value for `x`).
`puts`	Usually used to push registers or values onto the stack.
`gets(x)`	Accesses the stack with an index of `x` relative to the stack pointer, and is frequently used to read registers from the stack.
`alter(x)`	Modifies the stack pointer by the immediate value `x`. `alter(x)` is useful for clearing parameters off the stack after an assembly routine call.
`save_reg`	Pushes all register file (R) registers onto the stack.
`restore_reg`	Pops all register file registers off the stack.