Register Classification

This section describes the ADSP-218x registers. Registers are listed in order of preferred allocation by the compiler.

Callee Preserved Registers ("Preserved")

Registers `I2`, `I3`, `I5`, `17`, and `M0` are *preserved*. A subroutine which uses any of these registers must save (preserve) and restore it.

Dedicated Registers

Certain registers have dedicated purposes and are not used for other things. Compiled code and libraries expect the dedicated registers to be correct.

Caller Save Registers ("Scratch")

All registers not preserved or dedicated are *scratch* registers. A subroutine may use a scratch register without having to save it.

Circular Buffer Length Registers

Registers `L0` through `L7` are the circular buffer length registers. The compiler assumes that these registers contain zero, which disables circular buffering; they *must* be set to zero when compiled code is executing, to avoid incorrect behavior. There is no restriction on the value of an `L` register when the corresponding `I` register has been reserved from compiler use.

See "-reserve register[,register...]" on page 1-38 for more information about reserving registers.

Mode Status (MSTAT) Register

The C runtime initializes the MSTAT register as part of the run-time header code. The compiler and run-time libraries assume to be running in these preset modes. If you change any of the modes listed in Table 1-9, ensure that they are reverted before calling C compiled functions or functions from the C run-time library. Failure to revert to the default modes may cause applications to fail when running.

Table 1-9. MSTAT Register Modes

Mode	Description	State
SEC_REG	Secondary Data Registers	disabled
BIT_REV	Bit-reversed address output	disabled
AR_SAT	ALU saturation mode	disabled
M_MODE	MAC result mode	Integer Mode, 16.0 format

Complete List of Registers

The following tables describe all of the registers for the ADSP-218x DSPs.

- Table 1-10 lists the data register's file registers
- Table 1-11 lists the DAG1 registers
- Table 1-12 lists the DAG2 registers

Table 1-10. Data Register File Registers

Register	Descriptuon	Notes
AX0	scratch	
AX1	scratch; single-word return	
AY0	scratch	

Table 1-10. Data Register File Registers (Cont'd)

Register	Descriptuon	Notes
AY1	scratch	Argument 2 for compatibility call
AR	scratch;	Argument 1 for compatibility call
AF	scratch	
MX0	scratch	
MX1	scratch	
MY0	scratch	
MY1	scratch	
MR1:0	scratch	
MR2	scratch	
MF	scratch	
SB	scratch	
SE	scratch	
SI	scratch	
SR1:0	scratch; double-word return	

Table 1-11. DAG1 Registers

Register	Descriptuon
I0	scratch
I1	scratch
I2	preserved
I3	preserved

Table 1-11. DAG1 Registers

Register	Descriptuon
M0	preserved
M1	dedicated: +1
M2	dedicated: 0
M3	scratch
L0-3	not used, must be zero

Table 1-12. List of DAG2 Registers

Register	Descriptuon
I4	dedicated: SP
I5	preserved
I6	scratch
I7	preserved
M4	dedicated: FP
M5	scratch
M6	dedicated: 0
M7	dedicated: -1
L4-7	not used, must be zero

C and Assembly Language Interface

This section describes how to call assembly language subroutines from within C programs, and how to call C functions from within assembly language programs. Before attempting to do either of these, be sure to familiarize yourself with the information about the C run-time model (including details about the stack, data types, and how arguments are handled) in "C Run-Time Model and Environment" on page 1-132.

This section contains:

Calling Assembly Subroutines from C Programs

Before calling an assembly language subroutine from a C program, create a prototype to define the arguments for the assembly language subroutine and the interface from the C program to the assembly language subroutine. Even though it is legal to use a function without a prototype in C, prototypes are a strongly recommended practice for good software engineering. When the prototype is omitted, the compiler cannot perform argument type checking and assumes that the return value is of type integer and uses K&R promotion rules instead of ANSI promotion rules.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. Scratch registers can be used within the assembly language program without worrying about their previous contents. If more room is needed (or an existing code is used) and you wish to use the preserved registers, you *must save* their contents and then *restore* those contents before returning.

In general, you should perform the following steps when writing C-callable assembly subroutines:

- Familiarize yourself with the general features of the C run-time model. This should include the general notion of a stack, how arguments are handled, and also the various data types and their sizes.
- Create an interface definition, or "prototype", so that the C program knows the name of your function and the types of its arguments. The prototype also determines how the arguments are passed.

  In C mode, the compiler allows you to use a function without a prototype. In this case, the compiler assumes that all the arguments, as they appear in the call, are of the proper type even though this may not be desired. The compiler also assumes that the return type is integer.
- The compiler normally prefaces the name of external entry points with an underscore. You can simply declare the function with an underscore as the compiler does. When using the function from assembly programs, you might want your function's name to be just as you write it. Then you will also need to tell the C compiler that it is an `asm` function, by placing `'extern "asm" {}'` around the prototype.
- The C run time determines that all function parameters are passed on the stack. A good way to observe and understand how arguments are passed is to write a dummy function in C and compile it using the `-save-temps` command-line switch (on page 1-39). The resulting compiler generated assembly file (`.s`) can then be viewed.

The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

```
// Sample file for exploring compiler interface...
// global variables assign arguments there just so
// we can track which registers were used
// (type of each variable corresponds to one of arguments)

int global_a;
float global_b;
int *  global_p;

// the function itself

int asmfunc(int a, float b, int * p, int d, int e) {
// do some assignments so that .s file will show where args are
   global_a = a;
   global_b = b;
   global_p = p;
   //value gets loaded into the return register
   return 12345;
}
```

When compiled with the `-save-temps` option set (see on page 1-39), this produces the following:

```
// PROCEDURE: _asmfunc
   .global _asmfunc;
_asmfunc:
      SI = DM(I4 +  4);
      I0 = SI ;
      AX1 = DM(I4 +  2);
      SI = DM(I4 +  1);
      AX0 = DM(I4 +  3);
      DM(_global_b) = AX1;
      DM(_global_a) = SI;
      DM(_global_b+1) = AX0;
```

```
        RTS (DB);
        AX1 = 12345;
        DM(_global_p) = I0;
_asmfunc.end
```

For a more complicated function, you might find it useful to follow the general run-time model, and use the run-time stack for local storage, etc. A simple C program, passed through the compiler, will provide a good template to build on. Alternatively, you may find it just as convenient to use local static storage for temporaries.

Calling C Routines from Assemby Programs

You may want to call a C-callable library and other functions from within an assembly language program. As discussed in "Calling Assembly Subroutines from C Programs" on page 1-148, you may want to create a test function to do this in C, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated*. The contents of the scratch registers may be changed without warning by the called C function. If the assembly language program needs the contents of any of those registers, you *must save* their contents before the call to the C function and then *restore* those contents after returning from the call.

Do *not* use the dedicated registers for other than their intended purpose; the compiler, libraries, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

Preserved registers can be used; their contents will not be changed by calling a C function. The function will always save and restore the contents of preserved registers if they are going to change.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. Explore how arguments are passed between an assembly language program and a function by writing a dummy function in C and compiling it with the `save temporary files` option (the `-save-temps` switch on page 1-39). By examining the contents of volatile global variables in `*.s` file, you can determine how the C function passes arguments, and then duplicate that argument setup process in the assembly language program.

The stack must be set up correctly before calling a C-callable function. If you call other functions, maintaining the basic stack model also facilitates the use of the debugger. The easiest way to do this is to define a C main program to initialize the run-time system; maintain the stack until it is needed by the C function being called from the assembly language program; and then continue to maintain that stack until it is needed to call back into C. However, make sure the dedicated registers are correct. You do not need to set the `FP` prior to the call; the caller's `FP` is never used by the recipient.

Using Mixed C/Assembly Support Macros

This section describes the C/Assembly interface support macros available via the `asm_sprt.h` system header file. Use these macros for interfacing assembly language modules with C functions.

Your software package includes a version of the `asm_sprt.h` file. Table 1-13 lists and the following section describes the macros.

Table 1-13. Interface Support Macros

`function_entry`	`exit`	`leaf_entry`	`leaf_exit`
`alter(x)`			
`save_reg`	`restore_reg`	`readsfirst(x)`	`readsnext`
`putsfirst`	`putsnext`	`getsfirst(x)`	`getsnext`

function_entry

The `function_entry` macro expands into the function prologue for non-leaf functions. This macro should be the first line of any non-leaf assembly routine.

exit

The `exit` macro expands into the function epilogue for non-leaf functions. This macro should be the last line of any non-leaf assembly routine. Exit is responsible for restoring the caller's stack and frame pointers and jumping to the return address.

leaf_entry

The `leaf_entry` macro expands into the function prologue for leaf functions. This macro should be the first line of any leaf assembly routine.

This macro is currently null, but should be used for future compatibility.

leaf_exit

The `leaf_exit` macro expands into the function epilogue for non-leaf functions. This macro should be the last line of any leaf assembly routine. `leaf_exit` is responsible for restoring the caller's stack and frame pointers and jumping to the return address.

alter(x)

The `alter` macro expands into an instruction that adjusts the stack pointer by adding the immediate value `x`. With a positive value for `x`, alter pops `x` words from the top of the stack. You could use alter to clear `x` number of parameters off the stack after a call.

save_reg

The preprocessor expands the `save_reg` macro into a series of assembly language commands that push the following registers (on the ADSP-218x architecture) onto the C run-time stack:

`AY0`, `AX0`, `AX1`, `MY0`, `MX0`, `MX1`, `MR1`, `MR0`, `SR1`, `SR0`, `I0`, `I1`, `M0`, `M3`, `I5`

restore_reg

The `restore_reg` macro expands into a series of instructions that pop the stored registers off of the C run-time stack.

readsfirst(register)

The preprocessor expands the `readsfirst` macro into a series of assembly language commands that read the value off the top of the stack, write the value to `register`, and set up for a read of the next stack entry with the `readsnext` macro. The `readsfirst` macro references the stack-pointer (`I4`) and might be used to read values that were placed on the stack using the `putsfirst` and `putsnext` macros.

register = readsnext

The preprocessor expands the `readsnext` macro into a series of assembly language commands. These commands continue the read process set up by the `readsfirst` macro by reading the next value off the top of the stack and writing it to `register`.

putsfirst = register

The preprocessor expands the `putsfirst` macro into a series of assembly language commands. These commands write the contents of `register` to the top of the stack and set up for a write of the next stack entry with the `putsnext` macro.

putsnext = register

The preprocessor expands the `putsnext` macro into a series of assembly language commands. These commands continue the write process set up by the `putsfirst` macro by writing the next `register` to the top of the stack.

getsfirst(register)

The preprocessor expands the `getsfirst` macro into a series of assembly language commands that read the value off the top of the stack, write the value to `register`, and set up for a read of the next stack entry with the `getsnext` macro. The preprocessor expands the `getsfirst` macro into a series of assembly language instructions that read a value from the top of a function frame, write the value to `register` and set up a read of the next value with `getsnext`. The `getsfirst` macro references the frame-pointer (`M4`) and would be used to read function parameters.

register = getsnext

The preprocessor expands the `getsnext` macro into a series of assembly language commands. These commands continue the read process set up by the `getsfirst` macro by reading the next value off the top of the stack and writing it to *register*.

Using Mixed C/Assembly Naming Conventions

It is necessary to be able to use C symbols (function or variable names) in assembly routines and use assembly symbols in C routines. This section describes how to name C and assembly symbols and how to use C and assembly symbols.

To name an assembly symbol that corresponds to a C symbol, add an underscore prefix to the C symbol name when declaring the symbol in assembly. For example, the C symbol `main` becomes the assembly symbol `_main`.

To use a C function or variable in your assembly routine, declare it as global in the C program and import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

To use an assembly function or variable in your C program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

Alternatively, the `cc218x` compiler provides an "`asm`" linkage specifier (used similarly to the "C" linkage specifier of C++), which when used, removes the need to add an underscore prefix to the symbol that is defined in assembly.

Table 1-14 shows the C/Assembly interface naming conventions.

Table 1-14. Naming Conventions for Symbols

In The C Program	In The Assembly Subroutine
`int c_var;` `/* declared global */`	`.extern _c_var;`
`void c_func();`	`.extern _c_func;`
`extern int asm_var;`	`.global _asm_var;`
`extern void asm_func();`	`.global _asm_func;` `_asm_func:`
`extern "asm" void asm_func();`	`.global asm_func;` `asm_func:`

Compatibility Call

The `cc218x` compiler in VisualDSP++ 3.5 produces code that is not fully compatible with the Release 6.1 run-time model. However, the new compiler is superior in many ways to the old one, and your programs will be faster, smaller, and more reliable after the C code is converted to the new system.

The cc218x compiler provides a compatibility call to enable usage of existing libraries and special-purpose assembly language subroutines with the new compiler. This feature is available with a small amount of source code modification by adding an 'extern "OldAsmCall" ' specification to the prototype in the source program, similar to what is done when calling between C and C++ source programs. There is no compiler option for compatibility calls.

This feature provides full compatibility with the following restrictions:

- You cannot mix old and new compiled modules
- Old code is not allowed to call into a new compiled module
- A procedure pointer from a new compiled module is not allowed as an argument to an old routine

Some programs may not have any declarations of external assembly language functions. This is not good programming practice and should be fixed.

The effect of the OldAsmCall declaration is as follows:

- Pass the first two arguments in registers AR and AY1.

The C run-time stack for compatibility calls is normally used to pass the third and subsequent parameters to a called function. This changes if either of the first two parameters is a multi-word parameter, in which case, it and all subsequent parameters are passed on the stack. Functions that take variable arguments (varargs functions) will have the last named parameter and subsequent parameters passed on the stack.

- Postpend an underscore onto the external name.
- Look in the AR register (instead of AX1) for a one-word return value.

The `OldAsmCall` extern declaration can encompass one or more prototypes that define external entry points, as shown in the following example.

```
extern "OldAsmCall" {
int libfn(int flag, int * a);
void resetmach(int idle);
}
```