ANALOG DEVICES Engineer To Engineer Note EE-120

Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Interfacing Assembly Language Programs to C

Contributed by David Levine

Writing code in a high-level language (such as C) makes it possible to quickly develop portable programs. However, there are times when you may want to use assembly language for part of an application. Assembly code may be needed for access to special instructions or system registers outside of the C domain, or may be selectively used to achieve higher performance. This note contains helpful steps a programmer can use to interface assembly code and C safely and efficiently.

How to write a subroutine in assembly language that can be called from C

1. Familiarize yourself with the general features of the C runtime model. This should include the general notion of a stack, how arguments are handled, and also the various data types and their sizes. This information can be found in the C compiler manual.

2. Create a C interface definition, or "prototype", so that the C program knows the name of your function and the types of its arguments. The prototype will also determine how the arguments are passed. [1]

3. The C 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). If you are using the function from assembler programs as well, 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's an asm function, via 'extern "asm" {}' around the prototype. Or you can use #pragma linkage_name before the prototype to specify the actual entry point name.[2]

4. A good way to determine how arguments will be passed is to write a dummy function in C and compile it with the "stop after compile" option set ("-S" command line option). Here's an example; the assignments to the global show where the arguments can be found upon entry to "asmfunc".

```
/* Sample file for exploring
compiler interface... */

 /*global  variables ... assign
arguments to them so that we can
track which registers were used.
There's a global corresponding in
type to each argument. */

   int global_a;
   float global_b;
   int *  global_p;

 /***** the function itself *****/

int asmfunc(int a, float b, int * p)
{
    /* do some assignments so .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;
```

[1] C does allow a function to be used without a prototype. In that case, C will assume that all the arguments, as they appear in the call, are the proper type, without conversions; this may not be what is desired! C will also assume that the return type is integer.

[2] This is being phased in, and may not be available in older versions of the tools. Check the documentation.

}

For the TigerSHARC processor, this produces the following (compiled -S -O). Note that TigerSHARC passes up to four arguments in registers; check the runtime model for your processor to see how many arguments are passed in registers and in which registers. (The optimizer has gratuitously reversed the order of the assignments.)

```
// PROCEDURE: asmfunc
          .global _asmfunc;
_asmfunc:
          J8 = j31 + 12345;;
          [j31 + _global_p] = J6;;
          [j31 + _global_b] = XR5;;
          cjmp (NP) (ABS);
          [j31 + _global_a] = J4;;
```

If arguments are on the stack, they will be addressed via an offset from the stack pointer or frame pointer.

5. For a simple function, this may be all the information you need. The body of function then performs the computation, sets up a return value, and returns to the caller.

    - **Some General Caveats:**

    IMPORTANT: The runtime model defines certain registers as "scratch" and others as "preserved" or "dedicated". The scratch registers are available for immediate use without concern. You may also use the preserved registers, if you need more room (or if you're working with existing code), but you **must** save them first and then restore them before returning.

    Dedicated registers should not be used for anything other than their intended purpose. The runtime model conventions apply not only to the compiler, but also to the libraries, interrupt routines, and the debugger. The interrupt routines in particular rely on having a stack available.

    IMPORTANT: The compiler assumes that the overall machine state does not change during execution. This is processor specific. If the machine has modes, like integer/fractional, and they are changed in the assembly function, then they must be reset to their original values before exiting from the function. If the machine performs circular buffering when certain registers are non-zero, those registers must not be left altered (unless suitable reservations have been made on the related registers). This is more of a concern on some processors than others are.

6. For a more complicated function, you might find it useful to follow the general runtime model completely, and use the runtime stack for local storage. Again, 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. If you call other functions, maintaining the basic stack model will facilitate use of the debugger.

How To Create An Assembly Language Program That Can Call A C Function

Calling "out" from assembly language to C is a bit more complicated than writing a C-callable assembly function, but it can be managed successfully. This kind of call is particularly useful for calling existing C-callable functions, most notably in the library.

1. Familiarity with the runtime model is essential.

2. A sample C program that calls the function in question will provide an excellent guide to argument setup.

(Again, the trick of using global variables helps clarify the essential code). Here's an example:

```
/* Sample file for exploring
compiler interface...  */

/* global  variables ... pass as
arguments just so we can more easily
track which registers are used.
 (type of each variable corresponds
to one of arguments) */

   int global_a;
   float global_b;
   int *  global_p;
   int global_ret;

/** the function prototype **/

int asmfunc(int a, float b, int * p);

/** the test call **/

void caller() {
  global_ret = asmfunc(global_a,
           global_b, global_p);
}
```

The relevant portion of the assembly language file produced by the TigerSHARC compiler contains the argument setup and call.

```
_caller:        // prologue omitted
    J6 = [j31 + _global_p];;
    J4 = [j31 + _global_a];;
    XR5 = [j31 + _global_b];;
    call _asmfunc; q[J27+4]=J27:24;
                   q[K27+4]=K27:24;;
    [j31 + _global_ret] = J8;;
```

Note that the runtime model on this machine specifies that two register save instructions be done as part of the call instruction. Other machines may have similar conventions: check.

3. To call a C function, the stack must be set up correctly. The easiest way to do this is to have a C main program which will initialize the runtime system; and then hold that stack until it's needed to call back into C. Again, an example program may provide the necessary template. Even a simple function has the code to set up and tear down a stack frame.

4. Note the division of registers between "scratch" and "preserved." You must assume that every scratch register will be modified when a C function is called. If there's something in a scratch register that you need after the call, you must save that register prior to making the call.

Conclusion:

It is often necessary to create assembly language routines that can be called from C programs or that can call functions written in C. The step by step instructions given in this note provide the information you'll need. If you become familiar with the run time model, and create the samples as directed, you'll find that interfacing C and assembly code is easy.