DSP in C++ : Calling Assembly Class Member Functions From C++

Submitted by DL 12/2000

When using a high-level language (i.e. C or C++) to develop DSP code, it is often desirable to hand-code time-critical sections in assembly code to achieve optimal performance. These assembly routines, however, must be callable from that higher-level language. When developing with C, this is a straightforward process (see the VisualDSP++ C Compiler Guide, page 2-58), but in C++, where we now have to deal with things like inheritance and name-mangling, it becomes much more complex from an assembly standpoint.

This application note presents a basic scheme to preserve the simplicity of performing standard C calls from a C++ environment.

The material requires a basic understanding of the C++ language and an object-oriented programming environment. For some good resources, see the **References Section** at the end of this application note.

How is the class information passed to class functions?

There are two major challenges that one meets when trying to write C++ callable assembly code, or more specifically, when one tries to write a member function in assembly.

The first problem is *name mangling*. Name mangling is an operation performed by the C++ compiler where it modifies the name of a member function to include information about its input and output parameters. There are a number of reasons why this happens but the primary reason is so the compiler can differentiate between different member functions with the same name. Member functions of a class may have the same name as long as the parameters they take are different. For example, in the class listing below, there are two constructors with the same name. The compiler will append a text string to the labels of each constructor to represent the parameters they take.

```
class cDelay {
   public:
       float *       Buffer;
       float *       Ptr;
       int           Length;
       float         Input, Output;

       Delay( float input );
       Delay( );
}
```

A function called **Delay** from the class **cDelay**, which takes a single float as a parameter and returns a single float, will receive the following label from the compiler:

_Delay__9cDelay_STq2dmFv

Hence the mangled names... The problem here is that unless you have some software to mangle your names for you, it is extremely tedious to manually determine what a mangled name is going to be based on its parameters. If you wish to interface a function to assembly, you must know what this mangled name will be. As we will see further on, there is a nice way to bypass this whole process.

Now that we understand that name mangling is something we want to avoid, if possible, let's address the second issue : how do we access the information in an instance of a class from a member function of that class. **When a function is called in C++, the first argument passed is the *this* pointer, or rather, the pointer to the top of the class structure.** This passage is transparent to the programmer (unless, of course, the programmer is trying to program an assembly language function, which we'll get to soon).

In order to review how function parameters are passed, we will first start with a C example. When a C function is called, the parameters are passed as follows:

1st parameter - R4 register
2nd parameter - R8 register
3rd parameter - R12 register
4th parameter - 1st location of stack
5th parameter - 2nd location of stack
…

For example, say we have a C function called **add** whose prototype might look something like this :

```
float add( float x, float y);
```

When we call **Add**, the **x** parameter is placed in the R4 register and the **y** parameter is placed in the R8 register.

From a C++ member function, the parameter passing is slightly different. Instead of the first parameter being passed in the R4 register, the this pointer is passed in the R4 register and any other parameters are passed in the same fashion as is true for C, starting in the R8 register :

***this* (ptr to instance) R4 register**
1st parameter - R8 register
2nd parameter - R12 register
3rd parameter - 1st location of stack
4th parameter - 2nd location of stack
…

Let's start with the following class as an example. In this example, the Add function will add a real and an imaginary component to its own Real and Imag variables. It will then return the real sum.

Cplx_float.h

```
Class Complex_Float {
   private :
       float  Real, Imag;
   public :
       Complex_Float();    // constructors
       Complex_Float( float real,
                      float imag );
       float Add( float real,
                  float imag );
};
```

If we declared a new instance of a Complex_Float and called the Add function, the parameters would be passed as follows :

R4 - pointer to the current instance of Complex_Float
R8 - *real* parameter
R12 - *imag* parameter

For both C and C++, the return argument of a function is always placed in the R0 register.

Now suppose we wanted to hand-code the complex Add routine that we just defined in assembly code.

Add_.asm

```
#include <asm_sprt.h>
.segment  /pm  seg_pmco;

// function name is arbitrary
_AsmAdd:
.global _AsmAdd;

       // r4 contains a pointer to the
       // instance
       i4  = r4;
```

```
        // fetch the Real and Imag values
        // from our instance
        r0 = dm(0,i4);       // Real
        r1 = dm(1,i4);       // Imag

        // the real and imag parameters
        // are in r8 and r12

        // add real and imag components
        // and then store back to memory
        r8 = r8 + r0;
        dm(0,i4) = r8, r12 = r12 + r1;
        dm(1,i4) = r12;

        // now we return the real part
        r0 = r8;
        leaf_exit;
.endseg;
```

At this point, we have an assembly language function called _AsmAdd which we now want to call directly from our member function, Add. Most importantly, we don't want to deal with name mangling. In order to do this, we need to make some slight changes to our class like so:

Cplx_float.h

```
class Complex_Float;

extern "asm" float _AddAsm(Complex_Float *,
float, float);

class Complex_Float {
   private :
       float  Real, Imag;
   public :
       Complex_Float();    // constructors
       Complex_Float( float real,
                      float imag );
       inline float Add( float real,
                   float imag ) {
              return _AddAsm(this,
                             real,
                             imag); }
};
```

What we are doing is using an inlined version of Add to call our assembly function. The Add function is gluing our assembly function and the member function together. Because the Add function has been declared as an inline function, the second order of indirection here is eliminated.

This brings us to the next question :

How is a class stored in memory?

A class is stored in the DSPs memory just like a structure - each element is stored contiguously and in the order in which it is declared. For each instance of a class, we have a block of data stored in the DSP's memory.

For example, let's start with a simple class to represent a primitive packet of data:

Datapacket.h

```
class Packet {
       int    bytes;
       int *  payload;
       int    checksum;
};
```

Without diving too deeply into the C++ language, let's declare these and initialize them the old-fashioned way (i.e., without constructors) and see what we end up with in memory.

Main.cpp

```
#include "datapacket.h"

// declare 2 instances of the data type,
// Packet

Packet packet0, packet1;

Main()
{

       // initialize packet0
       packet0.bytes = 10;
       packet0.payload = new int [10];
       packet0.checksum = 0x12;

       // initialize packet1
       packet1.bytes = 5;
       packet1.payload = new int [5];
       packet1.checksim = 0xAE;

}
```

Each instance of Packet (i.e., packet0 and packet1) require 3 words of memory and will be placed right next to each other in memory since

we declared them consecutively. Let's say, for example, that the linker placed these two instances beginning at location 0x31000. After running through main(), the memory will look like this:

Memory location	Contents
0x31000	10 <- packet0
0x31001	0x34000
0x31002	0x12
0x31003	5 <- packet1
0x31004	0x3400A
0x31005	0xAE

The location of class data is independent of the declaration types, public, private or protected. Class data will be stored in the order in which it is declared.

If a class contains an array, the array data will be stored within the class structure as well. So, let's revisit that last example in a slightly different light:

Datapacket.h

```
class Packet {
       int    bytes;
       int    payload[5];
       int    checksum;
};
```

In this case, we have changed the **payload** variable to be a static array, which we will no longer need to dynamically allocate space for. This now means that the first memory location of our instance will contain the **bytes** variable; the next 5 locations will contain the values in **payload**; and the last location will contain the **checksum** variable. Instead of taking up 3 words of memory, each instance of this class now takes up 7 words of memory. Of course, we used memory from our heap when we dynamically allocated space for the **payload** variable, which required us to not only reserve space for those elements but a pointer to those elements as well.

If a class contains another class or a structure, that data will be inserted into the class as well at the location of the declaration.

What about inherited data!?

If a class receives inherited variables from a parent class, the parent class data comes first in memory. Let's look at an example:

"cfir.h"

```
class Filter {
       int type;
       float gain;
       };

// the cFIR class inherits from Filter
class cFIR : public cFilter {

   private:
       float *       Coeffs;
       float *       Delay_Line;
       float *       Delay_Line_Ptr;

   public:
       int           Taps;
       float         Input, Output;

       // constructor
       cFIR( float * coeffs,
             float * delay_line,
             int taps,
             int type,
             float gain );

       // destructor
       ~cFIR();
};
```

In this example, we have the parent class, **Filter**, which contains 2 variables – **type** and **gain**. Below that, a child class has been declared, **cFIR**, which inherits all of its parent's (**Filter**) variables as well. Again, the fact that some of these have been declared as private and others as public has no impact on where these will end up in memory.

We can now create an instance of a cFIR in our main code like so:

```
#include "cFIR.h"
```

```
cFIR * my_filter;

float coeffs[33] = {
#include "my_coeffs.dat"
};

float delay[33];

main()
{
       my_filter = new cFIR(coeffs,
                            delay,
                            33,
                            1,
                            1.0 );
}
```

In the example above, we used a constructor to initialize the data rather than doing it manually like in the previous example. So, let's look at how the data in my_filter is being stored. When we perform the dynamic allocation to create an instance of cFIR, the instance will end up in heap memory where all dynamic memory allocation needs are served. Let's look at this data in a slightly different fashion.

Memory Offset	Value
0x0	**Type** (inherited)
0x1	**Gain** (inherited)
0x2	**Coeffs**
0x3	**Delay_Line**
0x4	**Delay_Line_Ptr**
0x5	**Input**
0x6	**Output**

The point here is that inherited data comes first. In addition, **class function pointers (like constructors and destructors) are not actually stored in the class – this information is mostly hardwired by the compiler during compilation.**

Conclusion

While the C++ environment tends to be more complex than its C counterpart, there are some nice tricks we can take advantage of to keep things relatively simple. Concepts like static member variables, virtual functions, polymorphism, and some of the other more advanced C++ techniques will be covered in the next application note of this series.

References

Schildt, Herbert: *C++: The Complete Reference*
1988, Osborne McGraw Hill
ISBN 0-07-882476-1

Visual DSP C Compiler Guide & Reference
2000, Analog Devices Inc.