

# Programming digital signal processors with high-level languages

By Robert Jan Ridder

*In this article Robert Jan lists the pros and cons of programming digital signal processors in high-level languages. Then he compares the code size and the speed of some example programs written in assembly language and in C. According to these measurements, the drawbacks in code size and execution speed are acceptable – especially with the use of some C language extensions. Robert Jan also shows that a C language program, when compiled by a well-designed C++ compiler, has little additional overhead, allowing for a smooth transition from C to C++. Finally, he shows that the use of some C++ constructs can actually improve the runtime performance of C code, and he discusses methods that programmers can use to avoid code performance and size degradation when recoding assembly language DSP application programs in both languages.*

The increasing popularity of DSP-based products as well as their short life cycle have created a lot of schedule pressures on DSP software developers. Hand-held telephones have only recently become ubiquitous, and the mass introduction of set-top boxes for video-on-demand has not yet taken place. In markets this immature, both silicon and software technologies are changing rapidly, as standards evolve.

Demands for new features and falling consumer prices often make it necessary to change the hardware drastically between product generations. Even when a new DSP is chosen from the same family as the previous DSP, it might not be entirely software compatible with the earlier version. The problem is even worse if the new DSP is from a different manufacturer. In this case porting the existing DSP software can account for a major portion of the work needed to develop the new product – especially if the DSP software is written in assembly language.

## Using C to develop DSP-based systems

DSP application software has traditionally been written in assembly language. However, the increasing complexity of DSP algorithms and the expanding functionality of DSP-based devices (such as hand-held telephones with web browsers and personal agendas) have encouraged the use of the C programming language.

Although DSP software written in C is not always as efficient as assembly language, writing in C offers several advantages that make it attractive, including:

- portability between hardware platforms
- protection of the investment in program code
- reduced development time

The primary *disadvantages* to using C are slower execution and larger code size. However, these disadvantages are now less important because the newer DSP chips have faster clock rates and larger on-chip memories.

## Using C++ to develop DSP-based systems

The next logical step in high-level language application development is to use an object-oriented language, such as C++ or Java. (C++ is often the more attractive choice because it is a superset of C.) The abstractions used in object-oriented languages make the code more portable and encourage code re-use.

However, some developers have been hesitant to make the transition from C to C++ because they lack experience with it, and they fear that it will add another layer of inefficiency to the compilation of the source code.

In this article we will explore the advantages and the disadvantages of using C and C++ to develop code for a DSP-based system. We will then look at ways to minimize the disadvantages.

## Problem definition

There are several steps in the process of designing a DSP-driven device that meets a need, such as the need to communicate with anyone in the world. Once such a problem has been identified, a solution is designed in the form of an algorithm. This algorithm must then be implemented in the form of a list of executable instructions.

There is a large gap between the abstract (and mathematically complex) algorithms that must be run on DSPs and the limited set of registers and machine language instructions that are used to implement them. This problem is aggravated by:

- real-time execution constraints
- the limited precision of calculations on fixed-point DSPs
- scaling issues

Because of the difficulty in translating complex DSP algorithms into efficient executable code, assembly language has been used almost exclusively for DSP programming.

### Writing DSP code in assembly language

An assembler translates assembly-language source code into machine-code instructions that a DSP can execute. In most cases this is a one-on-one translation – each line of assembly language is translated into a single DSP instruction.

### Advantages to writing DSP code in assembly language

There are several reasons why a DSP programmer might use assembly language to program a DSP:

- Assembly language programming gives the programmer complete control, allowing for optimization of both execution speed and memory use.
- Performance requirements typically demand that DSP programmers squeeze every bit of performance out of their DSPs and, for small pieces of code, the best way to achieve that is to program the chip in assembly language.
- Even if optimal performance is not required, any opportunity to decrease the clock frequency is welcomed in portable applications, where it will prolong battery life.
- Cost savings can be obtained by careful programming in assembly language, if this permits execution of a DSP algorithm entirely within limited on-chip memory. (Off-chip memory is expensive because it must have a very short access time.)
- Execution speed can be increased if all the execution can be done from internal memory, thus avoiding the slowdown experienced when accessing off-chip memory.

However, programming at the assembly language level is very tedious. If this process could be automated, the DSP engineer could concentrate on the higher-level tasks. One way to do that would be to program in the C language. But there are some disadvantages to writing DSP code in C.

### Disadvantages to writing DSP code in C

There are several reasons why a DSP programmer might want to *avoid* the use of C to program a DSP:

- Older DSP architectures which were designed for assembly language are not very well suited to being programmed in a compiled language. For example, many high-level languages require random access to items on the system stack, and some DSP architectures do not provide instructions specifically for that purpose.
- Because older DSPs have typically been programmed in assembly language, there has not been a very large C compiler market for these DSPs. As a result, compiler vendors have not been motivated to put much effort into improving their compilers, and the compiled code has been relatively large and slow.

### Disadvantages to writing DSP code in assembly language

Based on the facts listed above, it might appear that assembly

language programming is the correct choice. However, there are also some really significant *disadvantages* to programming a DSP in assembly language:

- Assembly language programming requires the programmer to code DSP algorithms at a *very* detailed level. Because some of these DSP algorithms are very complex, the assembly language programmer might not see the opportunities for global optimizations as readily as a C programmer, who views the code from a much higher conceptual level. The result might be a relatively large executable code size.
- Programming an algorithm in assembly language typically takes much longer than programming the same algorithm in C. Algorithmic errors are more difficult to see, and debugging is more difficult. All of this results in lower programmer productivity and longer development times. Programmer productivity is particularly important because skilled programmers are scarce.

### Improving programmer productivity

Several methods have been used to improve the productivity of assembly language programmers, including:

- assembly language converter programs
- the construction of “re-usable” assembly language “building blocks”
- the use of C language compilers

### Assembly language converter programs

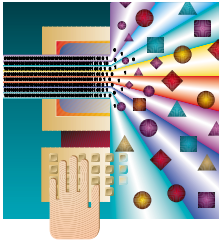
To help their programmers migrate between different DSPs (with their different assembly languages) some software developers have created *assembly language converter programs*. These programs translate the assembly language source code for one DSP into source code for another DSP – generally on an instruction-for-instruction basis.

The assembly language source code produced by these translators is typically less efficient than the input source code. In fact, these inefficiencies could easily cost more (in terms of performance) than is gained by migrating to a faster processor. In order to obtain some advantage from the migration, *manual-optimization* is typically needed. Unfortunately, the effort required to achieve the desired result is often judged too high, and a different approach is eventually chosen. (In many cases the only successful use of these converter programs has been for the application code on which they were developed and optimized.)

### Using “re-usable” assembly language “building blocks”

Some assembly language developers have attempted to implement small, re-usable building blocks, in the form of *macro libraries*. On a larger scale, some have attempted to implement entire DSP algorithms (consisting of thousands of lines of assembly language) as re-usable modules.

However, as mentioned earlier, assembly language varies from DSP to DSP, and even within a single DSP family.



Even when assembly language is common within a particular DSP family, re-use is not successful unless higher-level programming conventions are established to specify:

- the parameter calling convention
- the method of stack access
- the handling of interrupts

Without such conventions tedious and error-prone conversion is needed before “re-usable” modules can be linked together within an application program.

While a software developer can establish such standards for in-house use, the lack of industry support for these standards makes it impractical to sell finished software modules to third parties. Thus, the re-use of assembly language has been limited (at best) to applications written within a single company.

### The problem of limited on-chip memory

It seems that fast memory has always been at a premium in DSP systems. Fast, on-chip memory must be used very efficiently, and program code must be carefully structured and positioned to fit within that memory. Execution speed (and, to a lesser extent code size) is a primary concern.

In an effort to provide fast execution in spite of limited memory, programmers have resorted to:

- data overlays
- code overlays

### Data overlays

*Data overlays* allow software modules that execute sequential steps in a DSP algorithm to make optimum use of the limited on-chip data memory. Each software module stores its temporary variables in a common area of on-chip memory, overwriting the temporary variables of its predecessor. (Its own temporary variable values are also overwritten by its successor.)

Optimum use of this technique can get rather complicated. The programmer must decide which variable values should be stored outside the common area (to preserve them to future use) and which variable values are not needed. In general, it is difficult to use data overlays when writing in a high-level language. However, the C language includes the *union* keyword, which can be very useful for data overlays.

### Code overlays

*Code overlays* can be used to further improve execution speed and to cope with limited on-chip program memory. As the algorithm progresses from each step to the next, a new code module is loaded into the on-chip memory. Depending on the size of the code modules, there are two basic techniques:

- ALL of the code for the next step is copied from slow external memory (such as RAM or ROM) into on-chip memory.

- only the time-critical (inner-loop) portions of the code for the next step are copied from external memory into on-chip memory – the bulk of the code is then executed from the slower external memory.

Special run-time code is needed to accomplish the code overlays at the appropriate points in the execution of the algorithm, and the execution of this code must not be affected by the overlay process. Since high-level languages have no provision for this kind of code, it is difficult to employ code overlays in software written in a high-level language.

### Three recent solutions

Recent attempts to improve programmer productivity have focused on three techniques:

- C language compilers
- Java compilers
- graphically-driven code generators

### C language compilers

Newer DSPs are more compiler friendly, making the high-level language compilation more practical. At the same time, with more programmers writing their DSP code in high-level languages, compiler vendors are improving the performance of their compilers.

Efficient compilers for the C and Ada languages are now available for DSPs. For some DSPs, compilers for the object-oriented form of the C language (C++) is also available. While this language is still not used very much, many compiler buyers want to be sure that it will be available for future projects.

### Java compilers

Lately the Java language has also been considered quite often for DSP programming. However, more seems to have been written *about* Java than *in* it.

### Graphically-driven code generators

Some vendors offer flow-oriented DSP programming environments, where signal processing building blocks are interconnected *graphically*, to define the desired algorithm. Once this block diagram has been constructed on the display, the programmer launches a code generator. The resulting output is typically C language source code, or assembly language source code consisting of locally optimized macros.

### DSP developers are migrating from assembly language to C

C has long been the high-level language of choice in embedded software development, and has now become the most popular high-level language for DSP programming, and is gradually replacing the various DSP assembly languages. In fact, for the latest DSPs (which include multiple execution units and exposed pipelines) C is the only viable programming method, as writing assembly language code for these chips is very complicated.

Research indicates that a good programmer can create about 200 lines of debugged code per day, regardless of the programming language used. As one line of C code is the

equivalent of many lines of assembly code, productivity increases proportionally. The productivity increase has been estimated at 3 to 10 times, depending on the type of DSP being programmed.

### Advantages to using high-level languages to program DSPs

High-level language programming provides several clear advantages over assembly language programming:

- High-level language development environments take over many of the housekeeping chores of programming. This improves the reliability of an application program – especially in areas where testing is difficult.
- High-level source language is easier to read and understand. This helps the programmer identify errors and eases the job of adding features to an existing application program.
- There are many more people who can maintain programs written in C than in any single DSP assembly language.
- High-level language programs have better defined call interfaces, making it is easier to create re-usable software components.

### So, why have high-level languages not been used more in the past?

With all these advantages, it might be difficult to see why high-level languages have not been used more by DSP programmers. In the past some assembly language developers realized the advantages of writing in a higher level language, such as C, and decided to write their own C language compilers. However, developing a C language compiler involves a lot of effort. To minimize that effort, most of these C language compilers were based on *retargetable* compilers, such as the GNU C compiler. Unfortunately, retargetable compilers have built-in compromises that limit their performance, and the resulting compilers produced rather disappointing compilation results, in terms of code size and execution speed.

Faster product development doesn't help if the product can't meet the requirements of the application, so retargetable compilers were useless for generating production-quality code. Their use was limited to application programs that were not tightly constrained by memory or real-time requirements.

### Estimating code degradation when migrating from assembly language to C

Before considering the use of the C language (instead of assembly language) a DSP programmer should determine what price will be paid, in terms of execution speed and code size. In doing this, it's important to ask the right questions. For example

**instead of asking...**

*“Is the compiler generated code as good as code written by a well-trained assembly level programmer?”*

**you should ask...**

*“Is the compiler-generated code adequate for my application?”*

There are several reasons why it is difficult for a C compiler to match the performance of an experienced assembly language programmer:

- The expressiveness of a high-level language often does not match that of assembly language.
- The ANSI standard dictates that C compilers must generate code that will work under all circumstances. In contrast, assembly language programmers often optimize their software based on implicit knowledge of their own program (such assumptions about the range of a function parameter) that are not exploitable by a C language compiler.
- Compiler code optimizations are often performed only to a limited extent, to avoid long compilation times.

Some of these limitations will be lifted as DSP compiler technology improves, and as more powerful host computers are used to compile the code. However, there will always be some overhead attributable to the compilation process. Currently 50 to 100 percent overhead is typical. Of course, there are extremes where compilers generate very poor code, or conversely reach 0% overhead for a certain piece of source code. Eventually we can expect compilers to generate code with 20 to 50 percent overhead.

Note: The overhead of C compilation becomes smaller when comparing larger pieces of code. That's because assembly language programmers tend to spend less time optimizing their code when they have a lot of code to write.

Most of us have learned that design usually involves trade-offs. It seems that there is a “Conservation Law of Trouble” that says you can solve one problem, but you will create another problem in doing so. This might lead us to think that the use of C to program DSPs will inevitably lead to a degradation of code quality. However, this need not be the case. A programmer can “pay the inevitable cost” of using C by purchasing more expensive compiler tools. The conservation law still holds – it's just that we have choices about how we pay the costs.

### Optimizing your C software

Before attempting to optimize your DSP application code, you should realize that the 80-20 rule applies. This rule states that 80% of the execution time is spent executing only 20% of the code. This means that 80% of a DSP application program can be written in C (with all of its productivity advantages) without paying a large price. In fact, the 80-20 rule implies that improvements in this 80% of the code could never yield more than a 20% speed improvement – even if its execution time was reduced to zero!

Once the frequently-executed portions of the code have been identified, the high-level code for those portions can be optimized to maximize performance. Selecting the best algorithm is always the most gratifying method of improvement, as it can provide improvements of orders of magnitude, instead of small percentages.

Improving the high-level code should first be done in a target-independent way. If that does not provide adequate improvements, the code can then be *tuned* for the specific DSP hardware in the target system – at the expense of decreased portability to other hardware.

### DSP language extensions

To take full advantage of DSP computational hardware, many C compilers define *language extensions* using additional keywords to allow the use of typical DSP hardware features, such as circular buffers and fractional data types.

Figures 1 and 2 compare the efficiency of standard C code with code that employs C language extensions that allow optimal use of the DSP's computational hardware. Figure 1 compares the execution efficiency (in terms of the number of clock cycles per loop) and Figure 2 compares the code size (in terms of the number of program words in the executable machine code). We see that overall execution speed improves dramatically when using all of the language extensions. Code size goes down as well, but this improvement is not so large.

### Using in-line assembly language

Improving the structure of the most frequently executed 20% of the application code might not always be enough. In this case it might be necessary to use in-line assembly language.

Even if you know in advance that a certain section of your code will need to be written in assembly language, it will be helpful to those who maintain your code in the future if you write your code in C first. You can then:

- compile the C code to produce assembly language
- optimize that assembly language
- insert the optimized assembly language as in-line code

**Figure 1**

	Integer	Integer + extensions	Fractional	Fract + extensions
Real updates	114	114	150	100
Complex updates	221	186	343	114
FIR filter	500	500	400	100
IIR filter	269	208	256	144

**Figure 2**

	Integer	Integer + extensions	Fractional	Fract + extensions
Real updates	107	107	107	100
Complex updates	157	129	165	112
FIR filter	140	140	129	143
IIR filter	194	176	163	125

You can then use a *conditional compilation directive* to switch between the C code and the assembly language versions of that routine. This will allow a programmer to compile your C code for a different DSP type in the future when the need arises, without having to reverse engineer your in-line assembly language.

### Migrating from C to C++

The C++ programming language was developed to support object-oriented programming. It has become the language of choice for programming large applications on today's business computers, where graphical user interfaces fit the object-oriented design paradigm very nicely. With the availability of faster microprocessors, it is now possible to take full advantage of C++ to provide better abstraction of the hardware using abstraction layers.

However, the use of C++ for DSP programming is much less widespread. The primary reasons for this are:

- suspected inefficiency of code generated by C++ compilers
- limited availability of C++ compilers for popular DSP devices

### The advantages of C++

The advantages of C++ over C are:

- the ability to provide a higher abstraction level, thus reducing development time
- mandatory modular structure, leading to better code design and expandability
- better encapsulation of modules, leading to better re-usability
- upwards compatibility with C, which allows for easy migration

### The disadvantages of C++

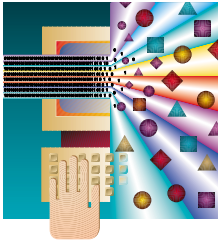
The disadvantages of C++ are:

- C++ language constructs can become complex
- C++ has a less intuitive program flow, due to the automatic calling of constructors and destructors
- most application programs (and many libraries) rely on `malloc()` and `free()`, so a heap is required
- more general constructs lead to greater memory requirements

### How current C++ compilers are constructed

Although some manufacturers do provide a direct C++ compiler, many C++ compilers are implemented as front-ends to ANSI C compilers. In compilers for embedded systems, the front-end approach prevails. We can thus expect to encounter all the limitations of the C compiler in the generated code, with additional limitations from the C++ front-end.

All C++ constructs can be directly translated to C constructs – with the exception of *templates*. For templates, the linker generates additional information about each template



instance, and then hands this back to the C compiler, so the C compiler can generate and compile the final C code. Since (for all practical cases) any construct in C++ could just as well be written in C, a programmer might question the need for a transition from C to C++.

Theoretically this is true. However, in practice it would be *very* difficult to consistently maintain the abstraction level and the complex C expressions. These tedious matters are better left to the C++ compiler, allowing the programmer to focus on the actual problem.

Note: The difference between C and assembly language could be described in a similar manner – anything that can be written in C could just as well be written in assembly language. However, writing assembly language takes much more time and is very device-specific.

The number of details that must be kept consistent is clearly shown by the difficulty of creating a *decompiler* – a tool that translates assembly language back into a C equivalent, or C into its C++ equivalent. Even if the input to the decompiler is restricted to compiled code, this is not a trivial task.

### Estimating code degradation when migrating from C to C++

Before using C++ (instead of C) a DSP programmer should ask what price will be paid, in terms of execution speed and code size. Ideally a C++ application program should produce code that is just as efficient as an equivalent C application program. In general, the designers of C++ compilers try to ensure that this is the case. Their effort can be summed up with the statement, “You shouldn’t have to pay for what you don’t use.” In principle, this means that:

- C data structures should not grow when compiled with a C++ compiler
- C program code should not yield more assembly code when compiled with a C++ compiler

Even when the programmer starts using C++ specific language constructs, the additional code required to support these constructs should be added *gracefully* to the generated code. That is, there should be no *large* increase of code size (or degradation of execution speed) just because a single C++ data structure (or coding method) has been added.

### Measuring C++ code bloat

To investigate how real the threat of code bloat is when using C++, we have compiled several well-known C benchmark programs with:

- a commercial C++ compiler (front end + underlying C compiler)
- its underlying C compiler only

Note: In running this experiment, we adapted the function prototypes of these (rather old) benchmark programs to the new C style, because the C++ compiler requires it – otherwise, we made no changes.

The values in Figure 3 show the additional overhead when using the TASKING DSP563xx C and C++ compilers to compile the C source code.

Figure 3’s results show some decrease of code quality, but not very much:

- the execution speed decreases by no more than 1.6%
- the code grows by no more than 6.8%

When using the C++ compiler there is a fixed increase (independent of the program size) of about:

- 220 words in code size
- 40 words in data size

This is caused by the startup code of this C++ compiler, which is included in the executable even if the code does not use C++ features. We conclude that the code size is increased by a nominal amount when moving from C to C++, and that the execution speed is hardly affected.

Of course the whole idea of migrating to C++ is to use at least *some* of the features of this object-oriented language. One of the things that may cause concern is the implicit class pointer (the *this* pointer) that is passed to every class member function. In an unfavorable situation, the object-oriented version of a program will have several unnecessary occurrences of this parameter passing.

To investigate the effects of this unnecessary overhead, a set of small programs was written with functions operating on a global structure. These programs included the following sections:

- init
- several actions (1, 3, or 10)
- exit

Figure 3

Benchmark	C compiler			C++ compiler		
	Speed	Code size	Data size	Speed	Code size	Data size
Whetstone	9076 kWhet/sec	4737	718	9066 kWhet/sec	4962	761
Dhrystone 1.1	55416 kDhry/sec	3946	7591	54771 kDhry/sec	4169	7633
Dhrystone 2.1	52590 kDhry/sec	4486	8598	51724 kDhry/sec	4709	8640
Sieve	41283 primes/sec	3198	8810	41283 primes/sec	3416	8852

The results when compiled with the C and C++ compilers are shown in Figure 4.

**Figure 4**

Program	C			C++		
	Speed	Code size	Data size	Speed	Code size	Data size
Small (1)	69	132	4	379	841	339
Larger (3)	97	150	4	407	857	339
Largest (10)	195	213	4	505	913	339

As we add more different action functions (in the “Larger” and “Largest” versions) the differences once again appear to be unrelated to the high-level program size. For all three version there is a C++ “overhead” of around:

- 700 program words
- 335 data words

In fact, the C++ version is a bit faster in executing the member functions than the C version, due to the efficient *this* pointer addressing.

The C/C++ difference shown in Figure 4 is larger than for the benchmark programs shown in Figure 3 due to the inclusion of C++ library code that calls the constructor and destructor. Another factor is the inclusion of the heap management functions `malloc()` and `free()`. Although these functions are not used in this program, the generic class constructor and destructor code includes them anyway. However, it is clear that these drawbacks become less important as the program grows larger.

The fact that object-oriented programming can lead to more efficient code can be shown with the following example. Suppose we need two functions that are identical, except for the inclusion of one additional block of code.

If we were writing this function in C, we would create a function with a *test* to conditionally execute this additional block of code, as follows:

```
void action(int type)
{
    /* base_code */
    if( type == TYPE2 )
    {
        /* additional_code */
    }
}
```

Of course one could also create two functions:

- one function for TYPE2
- one function for other types

However, the duplicated base code would be harder to maintain, since any bug found in one copy of the base code must also be fixed in the other copy of the same base code.

If we were writing this function in C++ (see Figure 5) we would:

- create a base class
- create two derived classes that have a virtual function.

In Figure 5 note that the version of function `action()` for class `Derived2` does not have a condition statement; calling it for the class at hand provides enough information. No code is generated for `Base::action()` if this function is never called for this class. The `base_code` is automatically inlined in the derived member functions, due to the declaration in the base class definition. (It could also have been declared `inline` to achieve this.)

So the content of `Derived2::action()` is actually the same as the C language version with the conditional block, and the content of `Derived1::action()` as the C version without it.

The only overhead involved is the regular C++ virtual function calling but, on the other hand, the function parameter type has disappeared. Thus, in many cases conditional code and switch statements can be avoided in C++ by the proper use of the class concept.

**Figure 5**

```
class Base
{
public:
    virtual void action(void) { /* base_code */ };
};

class Derived1 : public Base
{
public:
    virtual void action(void); /* defaults to Base::action(); */
};

class Derived2 : public Base
{
public:
    virtual void action(void) { Base::action(); /* additional_code */ };
};
```

## Optimizing your C++ software

All of the methods mentioned earlier for improving C code (when migrating from assembly language to C) are also useful for improving C++ code, provided they are supported in the C++ front-end. The use of language extensions and inline assembly in DSP programming often require the compiler vendor to make changes to generic C++ front-ends. In any case one can always resort to mixed C and C++ programming – or even add some inline assembly routines.

Some C++ features are not very efficient in an embedded environment, and should be avoided. This has given rise to sub-standards such as EC++ (Embedded C++) which forbids the use of some C++ language constructs.

Class libraries that have been written for desktop systems are also typically too inefficient for use in DSP systems. It is very helpful to have some knowledge of the underlying mechanisms of C++ constructs before employing them.

## Using a C code optimizer

Another way to improve the final code quality is to use an *optimizer* to improve the intermediate C code that is generated by the C++ front-end. Several commercial packages are available to transform the code generated by specific front-ends for a specific target DSP. Of course the effectiveness of an optimizer varies with the application, as well as with the programming style.

## The underlying C compiler must be optimized for C++ code

To reduce the code size of generated C++ code, the underlying C compiler should be optimized for compilation of the C code generated by the C++ front end.

One example is the use of derived classes, which are a core concept in object-oriented programming. Data members of base classes are placed in a data substructure that is offset from the address of the derived class. To call a function in the base class, the address of this data substructure must be passed to the base class function as the *this* pointer. It is therefore very important that the C compiler can handle offsets in nested data structures efficiently, so it can generate optimal code for this frequently used C++ construction.

As a second example, calling a virtual function translates to looking up a function pointer in a global table. (Note that, for data efficiency, this pointer is not stored within the class structure). A C compiler must be optimized to handle this construct (which is rarely used in regular C code) efficiently.

## Conclusion

The short product lifetime and the increasing complexity of DSP application software have made a transition from assembly languages to high-level languages inevitable. However, the performance difference between machine code generated from handcrafted assembly language code or from high-level language source code remains a concern. This article has shown that careful C programming techniques (when combined with DSP language extensions) can reduce performance degradation to an acceptable level. The availability of good development tools that support these language extensions also plays an important role in the development process.  $\Omega$



**Robert Jan Ridder** is a DSP Product Manager for TASKING, where he has developed an optimizing C and C++ compiler for the Motorola DSP56xxx line. He holds a Masters degree in Chemical Engineering, and has worked on embedded systems for many years.

He has been a music and computer enthusiast for almost two decades, and is a member of the Audio Engineering Society. He has four years of experience in designing and implementing real-time DSP audio systems – most importantly an artificial reverb system for theaters.

***TASKING, The Embedded Communications Company, brings together the software technology needed to compete in the embedded communications era. TASKING's award-winning integrated development environment, compiler, debugger, embedded Internet and RTOS products support a wide range of DSPs, 8-bit, 16-bit and 32-bit microprocessors, and microcontrollers for all areas of embedded communications. TASKING, founded in 1974, is a privately held company with headquarters in Dedham, Massachusetts, and engineering, sales and support offices in San Jose, California, the Netherlands, Germany, Italy, Japan and the UK. TASKING's 100,000 licensed users include the world's leading telecom, datacom, wireless and peripheral manufacturers.***

# A sample program showing how C can be used to program a DSP

To demonstrate the practicality of using C for DSP programming, TASKING provides an example DSP application program with its DSP56xxx compilers. This application program:

- calculates the power spectrum of a stereo audio signal
- displays the result on an oscilloscope

The power spectrum display is updated at a rate of 90 frames/sec, showing that this application program is practical for use in a real-world product. The program is also fully configurable at runtime. (For example, the size of the FFT can be specified at runtime). Portability has been demonstrated by its ability to run on several different evaluation boards, and even on DSP chips from different processor families.

Most of the source code for this application (including the input data windowing and the calculation of the dB values) is written in C, with the DSP extensions mentioned earlier. Assembly language is used only for the most time-critical parts, such as the FFT calculation and the interrupt routines. In spite of this, the program performs on a par with an assembly language implementation of the same algorithm.

Want to download the source code for this application program? If so, go to:

<http://www.tasking.com/technology/sdt-papers.html>

**If you have questions about this article, or if you would like more information about TASKING's products you can contact them at:**

**TASKING, Inc.**

333 Elm Street • Dedham, MA 02026

Tel: 781-320-9400 • Fax: 781-320-9212

Email: [dspengineering@tasking.com](mailto:dspengineering@tasking.com)

Web: [www.tasking.com](http://www.tasking.com)

Web: [www.embeddedcommunications.com](http://www.embeddedcommunications.com). (You'll need Macromedia Flash Player for this.)

**Trademarks**

TASKING and The Embedded Communications Company are trademarks of TASKING, Inc.