



Programming DSPs in High-Level Languages

ACE Associated Compiler Experts by

Version: 1.5
Date: May 23, 2003
Status: release
Confidentiality: public
Reference: CoSy-8070-dspprogramming

Programming DSPs in High-Level Languages

by ACE Associated Compiler Experts bv.

© Copyright 1999-2000 by ACE Associated Compiler Experts bv,
Amsterdam, the Netherlands.

All rights reserved. No part of this document may be copied, photocopied, reproduced or translated in any way, without prior written consent of ACE Associated Compiler Experts bv.

Every care has been taken in manufacturing the supplied product and its documentation. ACE Associated Compiler Experts bv will neither assume responsibility for any damages caused by the use of its products, nor accept warranty or update claims, unless stated explicitly otherwise in a special agreement.

The information contained in this document is subject to change without notice.

Printed in the Netherlands, May 23, 2003.

Many of the designations used by manufacturers and vendors to distinguish their product are trademarks. ACE Associated Compiler Experts bv has made every attempt to supply trademark information of manufacturers and their products mentioned in this document. ACE Associated Compiler Experts bv also recognizes any trademarks used in this document but not mentioned below.

Trademark notices

CoSy[®] is an international trademark of ACE Associated Computer Experts bv.

SuperTest[™] is an international trademark of ACE Associated Computer Experts bv.

HP-UX[®] is an international trademark of Hewlett-Packard Company.

Microsoft[®], Windows[®], and Windows NT[®] are international trademarks
of Microsoft Corporation.

Solaris[®] is an international trademark of Sun Microsystems, Inc.

Linux[®] is an international trademark of Linus Torvalds.

Intel[®], Pentium[®] and Xeon[®] are international trademarks of Intel Corporation.

UNIX[®] is an international trademark of Unix System Laboratories.

Red Hat[®] is an international trademark of Red Hat, Inc.

PostScript[®], Acrobat Reader[®], Acrobat logo[®] and PDF[®] are
international trademarks of Adobe Systems, Inc.

Netscape[®] and Netscape Navigator[®] are international trademarks
of Netscape Communications Corporation.

Motorola[®] is an international trademark of Motorola, Inc.

PowerPC[®] is an international trademark of International Business Machines Corporation.

SPARC[®] is an international trademark of SPARC International, Inc.

Programming DSPs in High-Level Languages

If you're thinking of programming your DSP in a high-level language instead of assembler, that high-level language will almost certainly be a C-type language. It's unlikely to be standard C, however, because standard C was never designed to handle the fixed point arithmetic, divided memory spaces and circular buffers that are typical of DSP architectures. Of the several C variants that are available to you, the one you choose will have a significant impact on both the portability and efficiency of the generated code.

With IP re-use now heralded as the only answer to shortening the time-to-market for new software developments, one thing is certain. In future, nearly all DSP software will have to be written in a high-level language to make it maintainable, re-usable and portable. Yet the high-level programming languages being adopted by the DSP community are virtually all based on C, a language that in standard form is not able to handle the fixed-point arithmetic, divided memory spaces and circular buffers that are typical of modern DSP architectures. In addition, the use of any such high-level language tends to decrease the efficiency of the generated code. Yet code efficiency is a critical requirement for many DSP programs, particularly those designed for real-time embedded systems.

So how do programmers utilize DSP features that the C language was never designed to handle, yet still achieve portability and highly efficient compact code? The answer is to use one of a number of variants of the C language, coupled with the use of advanced simulation tools and compilers. The relative efficiency of these C-language variants depends on how easily they allow programmers to reach the unique features of DSPs at the C language level.

C++ classes implement DSP features

One way of coping with the problem is to program your DSP in C++, which allows you to define DSP-oriented classes such as a fixed-point data type, circular pointer and circular array. Once these classes have been defined, you can use them to create corresponding objects that are manipulated using the language's standard operators (+, -, *, /, =, ==, !=, <, > etc.) in a class-specific way. As a result, common operations on these DSP-oriented objects are given the same natural, intuitive, feel as operations on C++ standard object types.

Because the definitions of new classes, and the way operators operate on them, are themselves written in C++, this approach has the advantage that your entire DSP program remains within the language. Using C++ with DSP classes therefore goes a long way to meeting the requirements of maintainability and re-usability. Programs written in this way can also be simulated and tested using standard software development tools and compiled using standard compilers.

However, the same reason that gives this approach the advantages of maintainability and re-usability is also the reason why it is difficult to get efficient DSP code from it. Operations on the DSP-oriented objects, which on the target processor typically execute in a single instruction cycle, appear in the C++ program as relatively long sections of code. C++ merely emulates these operations according to the newly defined DSP class rules, with the emulation

actually being written in C++. As a result, it is extremely difficult for compilers to recognize those parts of the C++ code that are DSP class operations and to map them onto appropriate target DSP instructions. Most compilers will simply treat the C++ emulation of a DSP class operation as standard C++, and generate much less efficient code from it. This compounds a problem that already exists with C++ compilers, in that because of the added complexity of the C++ language, these compilers already tend to generate less efficient code than standard C compilers.

Efficient mapping to target system instructions can only be done if the C++ code somehow contains ‘hints’ to the compiler that certain sections of code relate to specific DSP instructions. Unfortunately, the addition of these hints not only moves the program outside the limits of standard C++, it also means that the C++ code needs to contain target-specific information, both of which have serious implications for portability. Another disadvantage of using C++, particularly if you are only familiar with programming your DSPs in assembler, is that C++ is a considerably more difficult language to learn than C.

Ultimately, compiler technology may well develop to the point where C++ compilers have sufficient in-built intelligence to automatically recognize DSP class extensions and map them appropriately, but at present that situation is some way off. Because class-specific operations are themselves coded in C++, for example, such a compiler would have to recognize sections of standard C++ code and map them onto the target in a non-standard way. Nevertheless, when such compilers do arrive, C++ with DSP classes will be one of the cleanest ways to program DSPs. The CoSy compiler development system from ACE Associated Compiler Experts (Amsterdam, The Netherlands), with its ability to implement sophisticated and intelligent optimization strategies and its internal support of DSP features, is likely to be one of the main platforms on which these compilers are developed.

Intrinsics offer code efficiency

If you’re prepared to sacrifice portability in return for code efficiency, you can program your DSP in standard C and add specific DSP functionality in the form of ‘intrinsics’. These intrinsics, which act like function calls, can be directly mapped by the compiler onto target DSP instructions. However, the use of intrinsics doesn’t guarantee maximum code efficiency, because the compiler does not always know enough about them to carry out processes such as global optimization. Their advantage over inserting sections of assembly language, however, is that they protect you from issues such as register allocation and instruction scheduling, which are still taken care of by the compiler.

Because intrinsics are necessarily target architecture dependent, this ‘C with intrinsics’ approach is the one most commonly found in compilers that are sold by silicon vendors to support their own proprietary DSP chip families. Each member of a particular DSP family normally has a similar architecture and instruction set, which means that it is relatively easy for the manufacturer to supply a compiler with a comprehensive library of intrinsics that will result in reasonably efficient code generation. For chip vendors, portability is not an issue.

For customers, however, with a great deal of IP bound up in their application programs, portability is a crucial issue if they want to maintain IP re-use throughout a migration path to different and more advanced DSP architectures. The C with intrinsics approach doesn't give them that portability.

Programmers find a compromise

In an attempt to do what is practical rather than what is ideal, many of today's programmers adopt a compromise solution. To maintain the portability demanded by company IP re-use policies, they write their DSP programs using C++ with classes, ignoring the fact that this might result in inefficient code if it were compiled by a standard C++ compiler. At the C++

Why choose C for DSPs?

The C language was originally developed in the early 70's for programming general-purpose computers such as the DEC PDP-11. Because of the diversity of the applications that were run on these computers, they utilized floating-point arithmetic and had large contiguous memory spaces — an architecture that is still used today in the ubiquitous PC.

Digital signal processors (DSPs), however, have evolved for a very special purpose — namely to process real-world analog signals in the digital domain, often with the additional requirement of having to do so in real time. In many cases, this means that their hardware architecture is specifically designed to speed the execution of one type of instruction — the multiply accumulate instructions needed to execute the Fourier series that lie at the heart of functions such as Finite Impulse Response (FIR) filters. To maximize the execution speed of this very specialized hardware architecture, DSPs therefore operate with fixed-point arithmetic and simultaneously fetch multiple operands from divided memory spaces.

Until very recently, DSPs were not programmed in high-level languages. They were programmed almost exclusively in assembler. Some were even hardwired to perform a single DSP function. With only a few hundred bytes of code required to implement the core of many DSP algorithms, the 'hand-crafting' required for these techniques was a practical proposition.

Today, however, two factors are forcing DSP programmers to use high-level languages. Firstly, many of today's more flexible DSPs are quite capable of performing irregular DSP algorithms, such as speech codecs, as well as the highly regular 'tight-loop' algorithms required for functions such as FIR filters. They can also be used to implement system control functions. This results in the need to write much larger programs, which is easier and quicker using a high-level language. Secondly, the short time-to-market requirements of very fast moving markets where DSPs are used extensively, such as in mobile telephony, demand that software is both re-usable and portable. The only way to achieve these two objectives is high-level language programming,

So why choose C as a programming language for DSPs? The answer is simple. It is easy to learn and intuitive to use, which means that it is already ubiquitous within the programming community. Undeniably, the standard C programming language doesn't support the fixed-point arithmetic, divided memory spaces and circular buffers that are typical of DSP architectures, but as the accompanying article illustrates these are not insurmountable problems.

Only a few short years ago, if you asked when we would see efficient C-compilers for DSPs, even the experts would say that they were at least a decade away. Today, if you don't have a C-compiler for your latest DSP chip, then you don't stand a chance in the marketplace. The changing environment in which programmers now have to work means that they demand it.

level they then test their programs on a workstation to check basic functionality, such as the integrity of their DSP algorithms.

Once they have got the basic functionality right, they then manually translate their C++ programs into C with intrinsics, often translating critical parts directly into assembler. This allows them to run the programs on a simulator to get closer to target system performance or even to compile the whole program into machine code so that it can be run in real-time on a hardware emulator. Although their programs have the appearance of being portable at the C++ level, in practice everything below this is non-portable. As a result, if a decision is made to change to a different DSP architecture, a great deal of work still needs to be done to prove the software on the new architectural platform. Even if portability is not a primary concern, having to maintain application programs that exist both in C++ and C with intrinsics, especially when manual translation between the two is required, is not conducive to good quality control.

DSP-C offers portability and code efficiency

On the principle that if the language doesn't fit the application then the language needs modifying, ACE Associated Compiler Experts, in close collaboration with the DSP industry, proposed extensions to the C language that give it the fixed-point data types, circular pointers and divided memory spaces required for effective DSP programming. Although this makes the new language, currently called DSP-C, slightly non-standard (a situation that may be corrected if the proposed extensions are adopted by the ISO standardization body), the important thing was that ACE ensured two things. Firstly that the new extensions enjoyed the same level of language support as existing C constructs, and secondly that they were implemented in such a way that compilers could easily recognize and handle them.

For example, in much the same way that C includes predefined routes for type conversion from a 'short' to an 'int' to a 'long', DSP-C provides similar type conversion routes from a 'short fixed', to a 'fixed' to a 'long fixed'. In other words, the new type definitions in DSP-C are supported in the same logical way that programmers have come to expect from a language like C. While the C with intrinsics approach tackles DSP issues from the target system viewpoint, resulting in a lack of portability, DSP-C tackles these issues from within the language itself. As a result DSP-C programs remain highly portable.

The compiler issue is just as important, and it is what distinguishes DSP-C from the C++ with classes approach. As mentioned earlier, compiler technology is not sufficiently advanced at the moment to allow C++ compilers to handle the necessary DSP classes and type conversions in a way that results in highly efficient target code. C compilers on the other hand are already very efficient, and with DSP-C extensions built into the language, this same efficiency can be carried over into the DSP domain — a fact that ACE has demonstrated with DSP-C compilers generated using the CoSy-DSP version of its CoSy compiler development system. With the ability to perform global optimizations that focus specifically on DSP operations, these compilers produce highly stable DSP code that is easier to debug, resulting in short time-to-market development cycles.

With many DSP programmers only now moving over from assembly language to high-level language programming, it is important that all the necessary tools are in place to make the process as easy and effective as possible. In addition to DSP-C and its associated compilers, ACE has also developed a DSP simulator that allows DSP-C programs to be emulated with ‘bit-true’ accuracy on standard workstations. This means that the effectiveness of different DSP algorithms and application programs can be evaluated early on in the design cycle, even before target system silicon or an instruction set simulator is available to run them.

While DSP-C has yet to become an official standard, it fulfills such an urgent need that it is already in use by a large number of DSP companies worldwide. Together with the availability of compilers for many of the latest DSPs, it is already making a significant impact on IP re-use and shortened time-to-market within the DSP industry. And with DSP-C now providing direct support for DSP-specific operations and architectures, it is logical to predict that the principles behind DSP-C will ultimately be transferable to the C++ domain, in much the same way that the C language itself developed into C++.