TASKING

Efficient Programming Techniques for Digital Signal Processing

Published: DSP Germany
Date: 7 and 8 October 1998

By Robert Jan Ridder
TASKING Software BV
Amersfoort, The Netherlands

Abstract: *In this paper some programming techniques are proposed to decrease the development time by using a high-level language without impairing the real-time performance and memory footprint too much. Reusability of code is also an issue that gets some attention. The discussion will focus on the use of C and C++ as high-level languages, and prepackaged software like real-time kernels and class libraries. Many of the methods discussed are equally applicable in a Java environment.*

Introduction

The use of high-level languages, prepackaged software and reuse of existing proprietary code has become a necessity in the DSP community. In general, this raises concern about deterioration of code density and execution speed. In this paper some very practical guidelines are given to limit this effect, that invariably occurs as a trade-off for development speed and maintainability. As the C programming language is the most popular high-level language for DSP applications, the discussion will focus on this language, with some sidesteps to its probable successors C++ and Java.

Project design

When a project is started it pays off to create a good design before delving into writing code. This avoids a lot of problems in the debugging phase when a poorly designed project has to be restructured to get it working. It also creates code that will prove to be more useful in succeeding projects.

After the external requirements of the project have become clear, a first inventory has to be made of the required functionality. A decision has to be made between what should be implemented in hardware and what should be done in software. At this point, one can decide if a real-time kernel and/or class library should be used. Although the complexity and the cost of a real-time kernel may seem excessive at first, the effort and cost behind a homegrown kernel should not be underestimated [1]. Class libraries for C++ or Java can create very efficient implementations for frequently used objects like lists, stacks and buffers. By using the C++ language feature of templates these can be created in optimal form from a single portable and well-tested source.

The interrupt priority structure and the partitioning of available on-chip hardware can then be decided upon. Many DSPs have very limited on-chip memory resources that must be shared between program parts in order to achieve the required performance. If program and/or data overlays will be used, they

must also be defined in this stage. Some documentation of these engineering decisions and their background can be very useful later on.

When starting to write the software, a top-down approach has a lot of benefits. If it is unclear whether some of the project requirements can be achieved, a pilot study of those parts should be done first. Creating header files and stub routines with an initial prototype provide a framework for the program. To improve portability, all definitions that depend on the corresponding processor should go into a separate module. All definitions that are specific for the project (e.g. sampling frequency, number of filter taps) should go into another header file.

Software that depends on a particular piece of hardware (e.g. a serial port) should go into a separate module on the next level. An abstraction can be made to separate the functionality from the hardware (e.g. read a byte from or write a byte to the serial port). In this way a serial port call becomes independent of the actual hardware used.

Every module should have a header file that defines what is externally accessible. This file should also be included in the module, to enable the compiler to check the prototypes with the definitions.

In order to improve reusability, dependencies between modules should be kept as few as possible. Header files should be listed in modules where necessary, first C library header files, then all project-wide header files, and finally module header files going from low level modules to high level modules.

Dependencies on hardware specifics should be limited to as few modules as possible. The memory layout should ideally be left to the linker, and not coded into the modules themselves.

These rules will lead to faster development, better portability and better reusability of the code, independent even of the programming language.

Generic efficient C programming

In this chapter some guidelines are given for writing C language programs that translate to efficient code. Many of these rules are a result of the C language definition as per the ANSI standard, and thus a compiler can not optimize their inefficient counterparts. Most of them are clear to compiler developers but unknown to many application developers. These guidelines are also applicable to C++ and Java, in that they help to massage the generated code in the desired direction without massive changes. They therefore form an important first step in creating an optimal application.

To achieve the most efficient DSP calculations, they must be performed in the native data type of the processor. This means (single-precision) float on a floating point DSP, (single-precision) fractional on a fractional DSP, int on a fixed point DSP. To improve portability, it makes sense to create defines for this data type. Of course, moving code from a floating point DSP to one of the other types may be quite difficult due to overflows in calculations that require additional scaling code.

A program that does not require floating point operations should be scrutinized on the use of float calculations, to avoid linking in large parts of the floating-point library. The map file of the executable will show whether floating-point routines are linked in. This may also arise as a result of calling library routines. The innocent and simple-looking printf() routine contains a formatter for floating point numbers, and will pull in large parts of the C and floating point library. For simple string output, the puts() routine is a more efficient alternative.

C provides automatic type promotion and will insert appropriate code that can be large and/or slow, especially when floating point types are involved. Conversions can also block compiler optimizations

on variables. To avoid this as much as possible the types of variables should be chosen consistently throughout the program.

A good working knowledge of the promotion rules can help to avoid unnecessary conversions and to maintain optimal accuracy. Consider the following example:

```
float fir(float *coef, float *data)
 {   int i;  double res = 0;

    for(i = 0;  i < NTAPS;  i++)
        res += coef[i] * data[i];
    return round(res);
}
```

The result of this routine will have less-than-optimal accuracy, as the multiplication is done in single-precision floating point; its result is cast to a double and added to the intermediate value in ‘res’, following the ANSI C promotion rules. To improve this situation, the function can be rewritten as:

```
float fir(float *coef, float *data)
{   int i;  double res = 0;

    for(i = 0;  i < NTAPS;  i++)
        res += (double)coef[i]
                       * data[i];
    return round(res);
}
```

Note that a single cast to double suffices to force the multiplication to double precision.

Unexpected promotions can also occur from float constants, which have double precision by definition. The ‘f’ modifier can be used to alleviate this: write `3.1416f` instead of `3.1416`.

Compilers can only optimize clusters of floating point constants if they are surrounded by braces. This is because the order of floating point evaluation can change overflow behavior. The constructs

```
#define TWOPI 2*3.1416
#define TWOPI (2*3.1416)
```

can therefore compile to different code, depending on the expression. The second will always be optimized to a single float constant.

Passing or returning structures and unions to a function, requires a copy of the structure that can cost a lot of CPU cycles. In most cases, passing or returning a pointer to a structure works just as well and is a lot faster. On the other hand, if a structure must be copied, the built-in structure copy provided by the compiler is probably more efficient than any user-written data copying loop. For the same reason, the C library memcpy() or strcpy() routines should be used for non-structure data.

The result of a condition statement is always 0 or 1, and as this is built into the compiler it will probably be coded efficiently. So it is better to write

```
err = val > 2;
```

than the equivalent

```
if(val > 2) err = 1; else err = 0;
```

Single-bit bitfields occur often in I/O programming and as semaphores. If a bitfield is declared as signed, some compilers consider it to be either 0 or –1, others to be either 0 or 1. To avoid confusion, they should be tested against zero only, so write:

```
if(!bitfield.bit0)  /* .. */;
if(bitfield.bit0)   /* .. */;
```

but avoid the ambiguity in

```
if(bitfield.bit0 == 1)  /* .. */;
```

Modern compilers can efficiently eliminate superfluous local variables, but the optimal creation of intermediate values and the caching of global variables are much harder problems. The programmer should therefore use extra local variables in cases where he recognizes situations that require them, to guide the compiler into coding an efficient version of the algorithm. Conversely, writing very compact and complex expressions will most likely not lead to tight assembly code. This is because many compiler optimizations are limited to simple cases and will be switched off for these situations. Writing simpler code also greatly benefits the maintainability of the project.

Conditional assignments can lead to better code than equivalent if-then-else constructions. If the left-hand expression has side effects, it is the only way to avoid doubling the code for it, short of using a temporary variable. So write

```
lexpr = (condition) ? 0xFE : 0xFD;
```

instead of

```
if(condition) lexpr = 0xFE;
         else lexpr = 0xFD;
```

especially if ‘lexpr’ is an expression containing function calls, pointers or array indices.

Switch statements can be coded with a jump table, a chain of immediate compares, or a combination of both. The table method requires a fixed number of cycles to execute, but may require a lot of space if the range of values is large. For small or sparsely populated number ranges compilers will create a chain of tests and jumps instead. In some cases it is wise to handle a single value separately. If a certain value is known to occur in the majority of cases, this can improve execution speed. If a single value far outside the range of the others is separated, this may enable the compiler to create a jump table.

```
if(retval != RET_OK)
{   switch(retval)
    { case RET_NAK:
           break;
           /* .. */
    }
}
```

Loops in the application are indispensable and often make up for most of the execution time. The C programming language provides three types of loop constructs: the for-loop, the do-while loop and the while-loop (loops created with the goto-statement are intentionally neglected in this paper). The task of the compiler is to map these constructs on the instruction set of the processor. As a minimum, the processor has a conditional and unconditional jump instruction. Most DSPs have a zero-overhead looping capability, which is generally implemented as an unconditional repeat of a range of instructions. Zero-overhead loops do not need the loop counter update, test and jump back instructions normally used in loops and thus speed up processing.

Below the surface, the compiler will rewrite any loop to a goto construction:

```
for(<init1>; <cond1>; <update1>)
      <body1>;
<init2>; while(<cond2>) <body2>;
<init3>; do <body3> while(<cond3>);
```

becomes

```
      <init1>
top1: if(!<cond1>) goto end1
      <body1>
      <update1>
      goto top1
end1:

      <init2>
top2: if(!<cond2>) goto end2
      <body2>
      goto top2
end2:
      <init3>
top3: <body3>
      if(<cond3>) goto top3
```

At the cost of an extra jump in the initialization phase, the first two loops can be rewritten, in a process called loop rotation, to the faster:

```
      <init>
      goto tst   ; extra jump
top:  <body>
      <update>   ; for-loop only
tst:  if(<cond>) goto top
```

And finally, to get even faster performance, the loop condition code can be duplicated to make the first jump more useful:

```
      <init>
      if(!<cond>) goto end
top:  <body>
      <update>   ; for-loop only
      if(<cond>) goto top
end:
```

Most DSPs have zero-overhead loops that can be written as a simple form of a for-loop in C:

```
for(cnt = 0; cnt < <limit>; cnt++)
            <body>;
```

where <limit> can be either a constant or a runtime calculated value. It is desirable that the compiler generates zero-overhead loops as much as possible. To do that, it must determine the loop iteration count from the intermediate code. The loop count can be written as an expression like:

```
<limit> = ( <endval> - <startval> )
              / <update_val>;
```

As can be readily seen, the initialization, update and end-condition of the loop must be found by the compiler to make a zero-overhead loop possible. Most compilers will not generate a zero-overhead loop if the expressions become too complex, if they contain side effects or if the loop variable is changed conditionally in the loop body. As a rule, the loop expressions should therefore be written as clear as possible. The for-statement shows all loop parameters, including the update statement, and should therefore be preferred.

If the loop condition is initially not met, the loop body must be executed at least once in the case of a do-while loop, whereas the loop body must be skipped in the other cases.

In cases where the loop count expression does not evaluate to a constant, the compiler has to insert a test for negative loop counts if the expression is signed, as the loop count is usually treated as an unsigned value. If the expression is unsigned, this test before the loop can be omitted if the hardware handles a loop count of zero correctly.

```
int limit;
/* code that calculates 'limit' */
if(limit <= 0)  goto end;
{  unsigned cnt;
   for(cnt = 0; cnt < limit; cnt++)
            <body>;
}
end:
```

Therefore the small function

```
long array_sum(int *ip, TYPE n)
{     long result = 0;  TYPE i;

      for(i = 0;  i < n;  i++)
            result += *ip++;
      return result;
}
```

will compile to smaller and faster code if TYPE is 'unsigned int' than if it is 'int', as the initial test is avoided.

Without zero-overhead loops, the same function, written as a while statement, will compile to less efficient code if written as

```
long array_sum(int *p, int n)
{     long result = 0;

      while(n--)
            result += *ip++;
      return result;
}

or

long array_sum(int *p, int n)
{     long result = 0;

      while(n-- > 0)
            result += *ip++;
      return result;
}
```


than as

```
long array_sum(int *p, int n)
{     long result = 0;

      while(--n >= 0)
            result += *ip++;
      return result;
}
```

In the first case, the compiler saves the current value, decrements the counter and tests the old value. In the second case, the compiler decrements the counter and checks if the current value is positive. Apart from not having to juggle around two values, the decrement action most likely sets the processor flags, so the test is automatically included. The only behavioral difference is for n = MIN_INT, where an overflow occurs in the first and third case that is not present in the second case, but this can almost always be ruled out as an impossible situation. As the result of overflows of signed numbers is undetermined according to the ANSI standard, this behavior should not be relied upon anyway.

Modern compilers will be able to handle simple loop forms optimally, but in more complex situations the pre-decrement should be preferred over the post-decrement. This is especially true when the result of the post-decrement instruction is tested, or assigned to a variable, because the right-hand side expression is evaluated first:

```
while(n--)  ; /* test 'old' value */
var = n--;  /* assign 'old' value */
```

In general, down counting is more efficient than up counting, because the decrement can serve as the test for completion as well. Also, compares to zero are more efficient than to other values.

```
      move  #limit,a
top:
      ; loop body
      sub   #1,a
      jne   top
```

is shorter and faster than

```
      move  #0,a
top:
      ; loop body
      add   #1,a
      cmp   #limit,a
      jne   top
```

Many compilers have the capability to inline functions, i.e. to expand the function code where the function is called instead of generating a subroutine call. In C++ this has become part of the programming language. Some compilers even do this automatically if certain conditions are met. Using inlining can greatly improve execution speed while maintaining modularity and readability.

The virtual function feature of C++ can be used to gain speed in an application in a very clear way. Calling a virtual function is actually a function pointer dereference. If the called function is exactly tailored for the task, the overhead is smaller than when several more general subroutines are called. Inlining of functions can help to create such a tailored function in an elegant way. The same can be accomplished in C, of course, but the handling of the function pointers can become very cumbersome.

It is important to think in terms of the generated code to avoid unnecessary overhead. As shown above, the expansion of a for-loop leads to the condition test being executed [loop count + 1] times. The following example has relatively poor runtime behavior due to this fact.

```
void strlower(char *s)
{  size_t i;

   for(i = 0; i < strlen(s); i++)
       s[i] = tolower(s[i]);
}
```

The function strlen() is called strlen(s) + 1 times, although it is clear to the programmer that the length of the string does not change during the loop. The compiler cannot optimize this situation to avoid the superfluous function calls. In general a function can return a different value on each call, and the loop body may have an effect on the function result as well. Note that even standard library functions are not exempt of this, because a user can replace them with a different implementation. Thus, the compiler cannot optimize based on the expected behavior of the strlen() function.

A good way to program this, without resorting to an intermediate variable, is:

```
void strlower(char *s)
{  size_t i;

   for(i = strlen(s)-1; i >= 0; i--)
       s[i] = tolower(s[i]);
}
```

Of course, a more efficient implementation of this function will avoid the strlen() function completely by writing this function as:

```
void strlower(char *s)
{
    while( *s )
        *s++ = tolower(*s);
}
```

DSP-specific programming techniques

The third step in creating an efficient DSP program is taking advantage of the hardware features of the processor and the software tools. Enabling an instruction cache and using a global memory allocation strategy, are simple ways to speed up an application. Switching on the optimization options of the tools is another easy way to get more performance. Many compilers switch off some optimizations to make the debugging process more intuitive for the user.

As high-level languages normally lack support for code overlays, a fixed program layout must be used, but this can decrease the performance considerably. Code overlays have to be designed carefully to avoid a lot of code-copying overhead, and can complicate the program code by introducing an implicit state in the process. Data overlays, on the other hand can be easily created. The stack, used for automatic variables, and the heap, used for dynamic allocation, are reused by nature. Global variables for different overlaid code parts can be placed in a union to conserve memory in a C compliant way:

```
union
{   struct
    {   int var1;
        float var2;
```

```
    } overlay1;
    struct
    {   float var1;
        double var2;
    } overlay2;
    /* .. */
} data;
```

Static data should be avoided if data memory is to be conserved with overlays.

Memory layout in general can be very important for the execution speed. In many cases, placing a floating-point library in fast internal memory can speed up calculations more than tweaking the calculation code would. The same rule applies to placing critical subroutines in fast internal memory. Going further into the realm of less-portable features, the use of Direct Memory Access and interrupts instead of polling can increase the throughput of a DSP algorithm a lot.

The use of language extensions can enable the compiler to create programs that approach the speed of a handwritten assembly version, as shown in previous papers [4,5]. The portability issue that arises here can be overcome by a special set of defines to change the extensions into ANSI compatible constructs for other platforms.

Final optimizations

Finally the application can be run in a profiler to see what code parts are determining the execution speed. In many cases this will point out weak spots in the program that can be easily rewritten to create portable and faster code. If the application still does not meet the requirements, the critical parts that were designated in the profiling phase must be examined more closely. Rewriting them in a way that suits the specific compiler or re-coding them in assembly language by hand are the last resorts then. Still there are ways to retain portability when using these methods, for instance by using conditional compilation.

Conclusions

The transition to a high level language does not imply that all precautions and checks that are done when writing assembly language become unnecessary. Creating efficient DSP code in a high level language requires knowledge of the working of the compiler and a lot of common sense. With these guidelines, and some study of the compiler output a portable application can be created in a fraction of the development time used for an assembly version.

References:

[1] Warren Webb, “Embedded operating-system software – build or buy,” EDN Europe, August 1998, pp. 43-52.
[2] ANSI C language specification, American National Standard X3.159-1989.
[3] Dominic Herity, “C++ in Embedded Systems,” Embedded Systems Programming Europe, June 1998, pp. 20-26.
[4] Robert Jan Ridder, “Trends in application programming for Digital Signal Processing,” Proceedings ICSPAT San Diego, 1997.
[5] Denise Ombres, “C and C++ extensions simplify fixed-point DSP programming,” EDN, October 1996.
[6] Adrian Freed, “Guidelines for signal processing applications in C,” C Users Journal v11, n9 (September 1993):85, Center for New Music and Audio Technologies, http://www.cnmat.berkeley.edu.

