BY GERARD VINK

If you want high-performance code for digital signal processors, native C is not the best choice. But it is possible to tune code through extensions to the language. The extensions supported by tools on the market fall into two camps and this article explores how the tools handle extensions and where the pitfalls lie.

Programming DSPs using C: efficiency and portability trade-offs

FRANÇAIS

Si vous recherchez un code performant pour les processeurs de signaux numériques, le langage C natif n'est pas le choix qui s'impose. Mais il est possible d'affiner ce code par l'ajout d'extensions au langage. Les extensions supportées par les outils disponibles sur le marché peuvent être rangées dans deux catégories ; cet article étudie la manière dont ces outils traitent les extensions et quels sont les pièges à éviter.

DEUTSCH

Wenn ein besonders leistungsfähiger Code für digitale Signalprozessoren gewünscht wird, stellt natives C nicht die beste Wahl dar. Es ist jedoch möglich, den Code über Erweiterungen dieser Programmiersprache zu optimieren. Die Erweiterungen, die von den am Markt befindlichen Tools unterstützt werden, können in zwei Kategorien unterteilt werden. Dieser Artikel untersucht, wie die Tools mit den Erweiterungen verfahren und wo die Fallstricke liegen.

Today most DSP software used in hand-held consumer applications is written in assembly. In other DSP application areas where power consumption and the application's footprint is of less interest, the trend is that control code is written in C and time-critical inner loops are coded in assembly.

In the microcontroller market, C gained favour over assembly-level programming for well-known reasons. It offered reduced development time, portability and reuse of application code and reduced maintenance cost. The trade-offs were a limited penalty on code size and run-time performance. As the complexity and size of DSP applications both continue to increase, the same reasons that motivated the move from assembly to C for microcontroller applications now apply to DSP.

There is, however, one major problem. Standard C, as defined in C9X, is neither designed nor suited to implement DSP applications. DSPs use many hardware architectural features to optimise system performance and cost. Examples include:

- Support of multiple on-chip and off-chip address spaces
- Special address modification types to implement circular buffers and optimise array indexing for FFT transformations
- Fixed-point arithmetic
- Accumulators with extended precision and saturation logic

Standard C supports none of the above listed features. DSP C compiler suppliers use two fundamentally different approaches to give the C programmer access to these hardware features. One group supports an extensive set of intrinsic[1] functions and pragmas[2]. The other group introduces C-language extensions such as memory space qualifiers, array and pointer qualifiers, fixed-point data types and additional keywords.

This article describes the basics of both approaches and explains the pros and cons with respect to the:

- Readability and maintainability of C code
- Quality of error checking that can be done by the compiler
- Effects on the debug process
- Portability of C code
- Effects on compiler technology and code generation

DSP-C language extensions are vendor specific, and no de-facto standard exists. However, generally shared best-practices are being created.

A formal specification of the DSP-C language extensions can be downloaded from www.tasking.com/specifications/DSP-C.

This DSP-C derivative, an extension to ISO/IEC IS 9899:1990 has been submitted to ISO/IEC as background material for new functionality that can be included in new versions of the C standard. This article describes the reasoning behind the introduction of a subset of the DSP-C language extensions in an informal 'explain by example' approach.

BY GERARD VINK

Architectural features of DSPs

DSP-specific architectural features are meant to allow high data throughput while keeping the device costs low. When we look at the historical enhancements of DSP, we can consider device architecture and device technology separately. The basic DSP structures were designed and well understood before the semiconductor technology was available to support them.

DSP performance increases yearly, due to improved device technology, whereas the basic architecture of DSP remains quite stable. The introduction of very long instruction word (VLIW) concepts to increase the level of instruction parallelism is one of the few recent shifts in device architecture. However, before we discuss DSP-C language extensions, we have to understand the differences in hardware architecture between DSPs and general-purpose processors. The most visible differences are in the memory system, ALU design and the address generation logic.

Typically, DSPs have very complex memory systems. Based upon the organisation of the memory system, computer architectures are classified as either Von Neumann or Harvard architectures. The Von Neumann architecture is used by most general-purpose processors, with the exception of a few microcontrollers, and does not have separate data memories. The Harvard architecture uses separate memory spaces for the program and the data. All early, and most current DSPs are based on the Harvard architecture. This is because it allows a program instruction and a data word access at the same time.

In real-time signal processing, the efficient flow of data into and out of the processor is critical. The data memory is often further subdivided into multiple memory spaces to allow access to multiple data words within one cycle. Memory can be located either on-chip or off-chip. Typically, high-speed on-chip memory is limited in size and is used to store instructions and data used by inner loops that must meet hard real-time and throughput criteria. Low-cost, relatively slow off-chip memory contains the instructions and data associated with control code to which less strict time constraints apply. In addition to the various data spaces, there is often a dedicated address space to which peripheral registers are mapped.

If you browse through the data books of DSP manufacturers, you will see a division between fixed-point and floating-point devices, and that the bulk of the processors fall in the fixed-point category. Fixed-point processors are often 16 or 24bit devices, whereas floating-point processors are usually 32bit devices. Fixed-point processors are popular because they are far cheaper than their floating-point counterparts and most end-user applications can be implemented using these devices. Low-cost fixed-point processors are generally priced at roughly 20% of the lowest cost floating-point device.

Many inner loops of DSP algorithms use calculations in the form A+=B[N]*C[N], otherwise known as a multiply-accumulate (MAC) operation. Note that two source data items have to be read simultaneously to execute this operation in one cycle. Although the data items B and C are typically single-precision (often 16 or 24bit in size), the intermediate value (A) is stored in a dual-precision accumulator (32 or 48bit) to avoid cumulative rounding errors when successive MACs are processed in a loop.

Typically, DSPs have a mechanism that protects against overflow. An additional number of so called guard bits and saturation logic is added to the accumulator. Guard bits provide additional precision and, if overflow still occurs, the saturation logic fills the accumulator to its maximum or minimum value, but prevents it from wrapping around through zero in the way that a conventional general-purpose processor would.

DSPs typically support different address modification types for computing addresses. In addition to standard linear addressing, modulo, and bit-reversed[3] addressing are supported. Modulo addressing is used to efficiently handle buffering of incoming and outgoing signals. In the digital world, a signal is a data stream that has to be stored in a buffer before and after it is processed by the DSP. Modulo addressing allows you to define a fixed size region of memory as a buffer. When the modulo-buffer boundary is reached, the address is wrapped-around without zero computation overhead. The hardware may place alignment restrictions on buffers that are indexed using modulo addressing to simplify the hardware.

The basic mathematical model for processing digital signals is based on the Z-transform (the digital version of the Laplace transform) and the fast Fourier

Forming fixed-point numbers

On a 16bit architecture, numbers are usually represented in what is known as Q15 format. The number following the Q represents the quantity of fractional bits.

This implies that in Q15 format, each number is represented by one sign bit, 15 fractional bits and no integer bits.

The rules of forming fixed-point numbers are easily understood by studying the Figure 1, in which some 4bit Q3 numbers are shown. Q3, for example, means that there is one sign bit, 3 fractional bits and no integer bits.

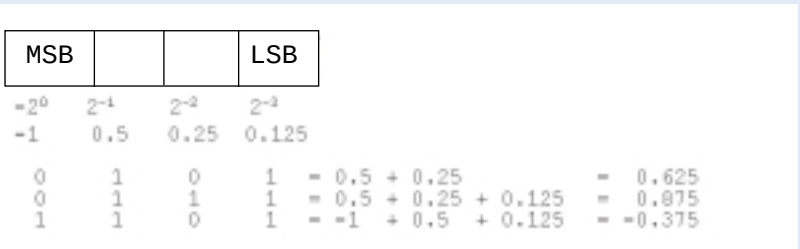

Figure 1: Format of Q3 fixed-point numbers

LISTING: 1

Fixed-point arithmetic using language extensions

```
fract _sat sf;
_fract f;

f  = 0.625r;
sf = 0.875r + f;   // sf=1 due to saturation
```

transform, which provides an efficient method to determine the frequency spectrum of any signal. An implementation of an FFT algorithm requires two data tables, one for the input values, the other for the output values. To obtain the output of, say, a four-point FFT in ascending order, the input values *input[N]* must be loaded in the following order:

N={0,2,1,3}.

Address calculations for a four-point FFT introduce minimal computational overhead, but this changes when computing a 1024-point FFT. The initial values are loaded in ascending order (linear addressing), and bit-reversed addressing is used to retrieve the values in the required shuffled order.

DSP architectures typically place alignment and size restrictions on buffers that are accessed via bit-reversed addressing. If a high-level programming language supports modulo and bit-reversed addressed buffers, then the compiler (in conjunction with the locator) should properly align the buffers in memory.

Assembly language versus C

Today, the majority of DSP applications that ship in high volume are written in assembly language. Programming in assembly language has many advantages. It allows you to implement optimised memory-usage patterns to avoid the need for off-chip memory or to reduce the costs of off-chip memory.

You can tune DSP algorithms towards the characteristics of any DSP architecture of choice, which allows you to use a cheaper derivative of the given architecture. In cases where power consumption is a concern, the structure of the assembly code can be adjusted towards this requirement. For example, to gain power reductions, you will typically avoid using external memory, keep variables in registers, use hardware loop constructs, and tune your operating clock frequency.

However, applications are increasing in size. Three years ago, a typical cellular phone contained mainly signal processing-related code, whereas modern cell phones now support email and Internet facilities. Assembly code cannot be ported to new chip architectures easily, which may be needed to fulfil new customer requirements in time. Generally speaking, it is more difficult to maintain and add new functionality to a program written in a low-level language.

VLIW-based DSP architectures with exposed pipelines are a challenge for the assembly programmer. Writing legal instruction schedules that fully exploit the parallelism offered by the hardware is a nightmare. Typically, these processors implement deep pipelines. As a result, registers and memory locations are updated a few cycles after an instruction is in the execute stage and dependencies between functional units limit the combination of legal operations in one instruction.

Because of the instruction-scheduling complications, VLIW processors are designed to be programmed using a high-level language. Often, compiler and hardware design are done in parallel, and the hardware design may be adjusted to bypass the restrictions in the available compiler optimisation technology.

ANSI C and DSP extensions

We have seen that there are many architectural differences between a general-purpose processor and a DSP. C was designed in the 1970s, before the first single-chip DSP was developed. Although C was meant to be used to implement applications that may interact with the underlying hardware, such as operating systems and device drivers, it lacks support for many of the DSP characteristics described above. The list of missing features includes support for:

- Multiple memory spaces
- Fixed-point data types and arithmetic
- Saturation
- Data types that map on accumulators with extended precision
- Circular buffers
- Bit reverse addressing

This section describes how the missing features described above can be supported at the C programming level, either by the introduction of C-language extensions or via intrinsics.

Fixed-point types and saturation

C9X defines the integral data types `char`, `short`, `int`, `long` and `long long` in signed and unsigned format, and the floating-point data types `float`, `double` and `long double`. No data types are defined to handle fixed-point arithmetic.

Various compiler vendors have solved this problem by introducing new data types: `short _fract`[4], `_fract`, and `long_fract` (see Listing 1). Unsigned fixed-point types are normally not defined, as these are of little use in DSP algorithms and are therefore not supported by the instruction set of any DSP. The unary operator ~ and the binary operators &, ^ and | are not allowed in fixed-point expressions. The `_sat` qualifier is introduced to specify that saturation must be applied whenever a result is stored in a variable of this type.

Note the new `r` suffix that is used in Listing 1. The `r` suffix specifies that the constants `0.625r` and `0.875r` are fixed-point numbers. Without the `r` suffix, the compiler would have assigned data type `double` to the constants

LISTING: 2

Fixed point arithmetic using intrinsic functions

```
int sf, f;
f  = 0x5000;       // bit pattern that coresponds to 0.625
sf = _sadd(0x7000, f);  // sf=1 due to saturation
```

0.625 and 0.875. This could result in additional type conversions and possible loss of accuracy.

The alternative way to implement fixed-point arithmetic is to store the data in a variable of an integral type and use intrinsic functions to perform operations in fixed-point format. This is shown in Listing 2.

Intrinsic function `int _sadd(int src1, int src2)` adds `src1` to `src2` and saturates the result. The panel on creating fixed-point numbers explains why the bit pattern of integer value 0x5000 (hexadecimal) equals 0.625 in fixed-point format.

To reduce complexity, the language extensions versus intrinsics approach will be compared on a one-by-one basis. First, we focus on readability of the code. Given the variable declarations in Listing 1, you know that expressions are evaluated using fixed-point arithmetic and that saturation is applied whenever a value is assigned to variable `sf`. In Listing 2, expressions are evaluated using integer arithmetic; however, the `_sadd` intrinsic specifies that fixed-point arithmetic is applied on variables of type integer and that saturation is applied when the result is returned. You have to read the whole function body before you know whether integer or fixed-point arithmetic is applied.

When the compiler parses the input file, it will search for errors and for suspicious constructs that could result in run-time errors and warn the programmer when necessary. Type information plays an important role in the error checking process. In Listing 2, the type information is not correct, as the compiler assumes that the variables are of type `integer`, whereas the programmer handles them as `fixed-point` types. The compiler therefore cannot perform error checks as rigorously as in Listing 1.

The debugger does not get the correct type information either. For instance, if you inspect the value of variable `f` in Listing 2, the debugger will show the value of integer `f` in decimal format and print `f=20480`. You have to remember that `f` represents fixed-point data and then convert the bit pattern to a fixed-point number.

During the compilation of Listing 1, the compiler can place the correct type information in the object file and the debugger can select the appropriate display format based on the type information. As a result, the value of `f` can be shown in the correct format `f=0.625`.

Suppose you want to prototype your source code on a PC, either under Microsoft Windows or Linux, using the native Visual C or gcc compiler. These compilers do not support the given language extensions, nor do they support the `_sadd` intrinsic function.

To compile Listing 1 on the PC, you have to create two defines. First, use `#define _fract float` to convert all fixed-point arithmetic to floating-point. Second, use `#define _sat` to let the pre-processor remove the `_sat` qualifier. Note that, due to these modifications, the simulation on the PC is accurate, but not bit-accurate.

If bit-accurate simulations are required, we can introduce more complex type definitions and macros. This way the readability of the code degrades to a quality-level equal to what is shown in Listing 2. However, the compiler and debugger will

LISTING: 3

Pointers to code and data space

```
char *rom = "in ROM"              // pointer to code space


char *ram = {'i','n',' ','R','A','M','\0'};
                          // pointer to data space

printf("%s, %s", rom, ram);
```

have correct type information, which will ease the debug process.

To simulate Listing 2 on the PC, you have to write code that implements the functionality offered by the _sadd intrinsic and all other intrinsics that you used (some DSP compilers offer over 100 intrinsic functions). This approach will be successful if you have carefully designed your code so that it will run bit-accurate simulations on a PC. For example, if you shift a fixed-point number by writing f=f<<1 instead of calling the appropriate intrinsic, your simulation will no longer be bit-accurate. In addition, bit accuracy is only achieved for fixed-point operations.

If you use a 24bit DSP, all integer arithmetic in your simulations on a host system will be done in 32bit precision instead of 24bit. Also, the simulation of floating point arithmetic will not be exact, as the floating-point library used by the DSP is typically optimised to achieve maximum run-time performance and will not be fully compliant with the IEEE754 floating-point format used on the PC.

Some compiler vendors offer a fixed-point data type that matches the size of the accumulator including the guard bits. This type is often named _accum. In production code, you do not want to copy the guard bits of the accumulator to memory, simply because it would slow down execution. However, if you are debugging the software, the compiler is often not allowed to apply all of its optimisations. Therefore, intermediate results may be copied to memory. However, if the guard bits are not copied to memory in optimised code, the debug version of the program may behave differently from the production version.

Memory Spaces

Most DSPs have separate memory spaces for code and data, and often the data space is further subdivided to enable the DSP to access multiple memory locations simultaneously. The instruction set typically supports a limited set of instructions and addressing modes to access data

LISTING: 4

FIR filter using C-language extensions

```
#define FIR_ORDER 20

_fract _X _circ    samples[FIR_ORDER];
_fract _Y          coeffs[FIR_ORDER];

/*                    N
 * Computes result = SUM( coef[i] * sample[T-i] )
 *                   i=0
 */
void fir_filter(int n)
{
   _fract _X _circ  *buffer_p = samples;
   long fract           result    = 0;
   int                  i;

   buffer_p += n;
   for(i = 0;  i < FIR_ORDER;  i++)
   {
      result += *buffer_p-- * (long _fract)coeffs[i];
   }
   result = _round(result);
}
```

stored in code space (ROM), whereas the data memory (RAM) is supported by a large set of instructions and addressing modes. To minimise the amount of RAM memory used by your application, it is desirable to locate constants and string literals in code space only.

In Listing 3, variable `rom` points to a string literal that can be located in code space. According to the C standard, a string literal may be located in read-only memory, but does it not have to be. Variable `ram` points to a character array located in data space. Again, according to the C standard, it shall be possible to assign new values to the initialised array. Therefore, the array must be located in read-write memory.

You see the problem here: two pointers are passed to the `printf()` function. As the pointers point to different memory spaces, different instructions are required to retrieve the data from memory. A generic pointer concept in which the memory space is encoded in the value of the pointer solves this problem. However, you have to pay for this elegant concept in terms of increased code size and data size, and reduced run-time performance.

When a pointer is de-referenced, first a test that identifies the memory space the pointer addresses is required, followed by a jump to the code fragment that contains the appropriate instructions to retrieve the data from memory. This overhead results in increased code size. The data size of a pointer also increases, as a pointer consists of two fields: the memory space and the offset within this space. An alternative solution is that all constant data is copied to data memory by the start-up code[5]. As a result, the constant data is in memory twice, so your application consumes more data memory than strictly necessary. However, compared to the generic pointer concept, the overall memory usage of the application decreases, as all data pointers (in contrast to function pointers) refer to data space only.

Memory-space qualifiers are used to qualify a pointer to one particular memory space. In the FIR filter shown in Listing 4, two arrays are defined.

The `samples` array is located in memory space _X, the `coeffs` array is located in memory space _Y. The advantage of locating the arrays in different memory spaces is that the DSP can access both spaces at the same time, minimising the processor stalls due to memory latencies.

If the compiler does not implement memory space qualifiers as shown in Listing 4, the compiler may provide a pragma for this purpose.

Alternatively, a pragma may be available to define new sections, and subsequently defined variables are located in the new section. In the locator description file, a section name is associated with a memory space.

Note that in the latter case, the compiler does not recognise that variables are located in, and pointers point to different memory spaces. This is a big disadvantage for the compiler optimisers. In this situation, the optimisers have no infor-

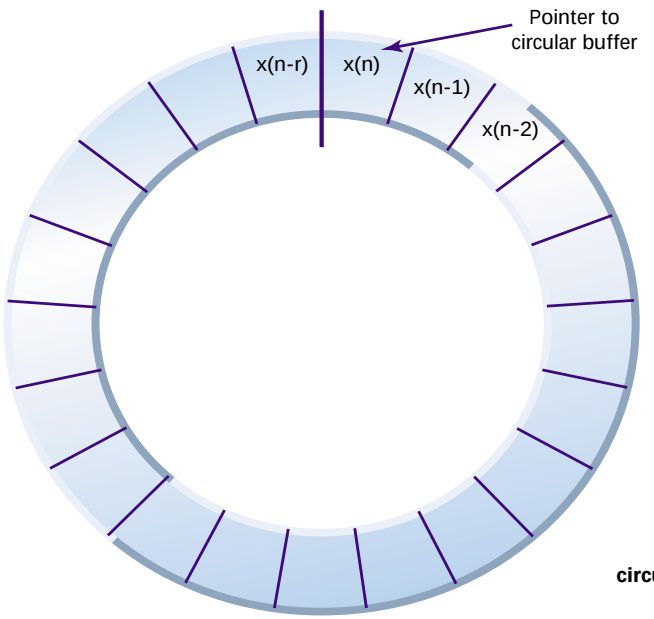

Figure 2: Logical structure of circular array or buffer

mation about memory-latency related dependencies between variables, which prevents the compiler from being able to generate an 'ideal' instruction schedule.

Circular buffers

Most DSP filter algorithms process long lists of data that must be multiplied in the correct order. Although the coefficient values are static, the input data changes every sample period (for example, the *x(n)* value for one sampling period becomes *x(n-1)* in the next, then *x(n-2)*, until it drops off the end of the delay chain). Circular buffers provide an efficient way to handle these input lists. New data that is placed in a circular buffer is placed one position above the previous sample.

If the buffer is filled to the top, the next sample will replace the first buffer location and so on. Typically, DSPs support modulo address modification to implement circular buffers with zero computation overhead.

ANSI C does not support circular addressing. So, many DSP compiler vendors have introduced the `_circ` pointer and array qualifier. It can be applied to one-dimensional arrays only.

In Listing 4, the input for the FIR filter is placed in array `sample`, which is defined as a circular buffer, represented in Figure 2. It is located in `_X` memory and contains 20 items of `_fract`. Argument `n` identifies the most recent entry in the `samples` buffer.

The first iteration of the for loop calculates `result += sample[n] * coeffs[0]`; the second iteration calculates `result += sample[n-1] * coeffs[1]`; and so on.

Finally, let's take a look at debugging aspects of circular buffers. Taking Listing 5 as an example, suppose you are single stepping through the loop and you are in the sixth iteration (that is `a=6`). What would you expect to see if you ask the debugger to show the value of `x[a]`?

Various answers are possible. First, 'array subscript out of bounds' is possible, as the array contains five items. Second, 'x[6]=42031' is also possible, as the debugger could read the data from outside of the array boundaries and retrieve the value of variable `dummy` instead.

LISTING: 5

Debugging aspects of circular buffers

```
int _circ x[5];
int _circ *p = x;
int  dummy = 42031;

for (int a=0; a < 7; a++)
{
   *p = a;   // x[0]=5; x[1]=6; x[2]=2; x[3]=3; x[4]=4;
}
```

Third, the most desirable answer is 'x[6]=6'. If the compiler passes the type information of `int _circ x[5]` to the debugger, the latter knows that variable `x` is of type *circular array of integer* with size of 5, and it therefore can deduce that it has to read the memory location associated with x[1]. The same reasoning as applied to circular buffers (that is, modulo address modification) can be applied to bit-reversed addressing.

Conclusions

Standard C does not allow the application programmer to exploit the capabilities of the DSP hardware. However, either introducing DSP-C language extensions or implementing intrinsic functions in the compiler can solve this problem. The author considers DSP-C language extensions to be the superior solution because of the following reasons:

- This improves the readability of the source code.
- The additional type information allows the compiler to perform thorough static error checks.
- The compiler optimizers can use the additional type information to fully exploit the address modification types supported by the DSP architecture and to create more aggressive instruction schedules.
- Debugging the source code will be easier. First, because the source code is easier to read and understand. Second, since the debugger has correct type information the value of variables are displayed in the correct format and when appropriate the debugger can apply modulo and bit-reversed address modifications.
- Porting the source code to a new DSP architecture will take less effort.

The author encourages the readers to re-implement Listing 4 without using any DSP-C language extensions. Next, you can use the pragmas and intrinsics offered by the DSP compiler of your choice to optimise the code. Afterwards, decide for yourself whether you agree with the above conclusions.

Gerard Vink studied mechanical engineering and computer science. He began his career developing CAD and computer graphics software. He has worked for Tasking since 1988. Before obtaining his current position as R&D manager, he was active as a consulting engineer and project manager.

Notes

[1] An intrinsic function is a function that is automatically in-lined by the compiler. The code associated with the intrinsic function is built into the compiler. In contrast to in-line assembly functions, intrinsic functions are inserted into the intermediate code representation used by the compiler and do not interfere with compiler optimizations.
[2] A pragma is a preprocessing directive of the form: #pragma pragma-token-list new-line and causes the compiler to behave in an implementation-defined manner. Pragmas can give directions to the code generator of the compiler.
[3] Also known as 'reversed-carry' addressing.
[4] Fract refers to fractional (or fixed-point) type which is a number in the range of [1,-1]
[5] The assembly code that is executed by the system before the main() function is invoked.