The Embedded C Extension to C

Marcel Beemster, Hans van Someren, Willem Wakker
ACE Associated Compiler Experts bv.[1]
{marcel,hvs,willem}@ace.nl

Class #463

Abstract

Embedded C is a language extension to C that is the subject of a technical report by the ISO working group on C named "Extensions for the programming language C to support embedded processors" [3]. It aims to provide portability and access to common performance increasing features of processors used in the domain of DSP and embedded processing.

Embedded C adds fixed-point data types, named address spaces and hardware I/O to C. Fixed-point primitives give the programmer direct access to a processor's fixed-point functionality. Named address spaces can be used to give control over multiple memory banks, which are typically present in DSP processors to increase the effective bandwidth to the ALU. The hardware I/O specification provides a standardized abstraction layer for accessing I/O hardware registers.

Fixed-point primitives and named address spaces are performance increasing features. They are motivated by a practical and economic need to program DSP processors in a high level language instead of assembly. The hardware I/O specification improves portability, allowing the same (driver) source code to run on multiple platforms.

Embedded C it is not part of the C language as such. The aim of the ISO report is to provide common ground for initiatives that currently exist within the industry to flourish. The report has been approved in 2004.

1. Motivation

High level language programming has been in use for a long time for embedded system development. However, assembly programming still prevails, particularly for DSP based systems. DSP processors are often programmed in assembly language by assembly programmers that know the processor architecture inside out. The key motivation for this practice is performance, despite the many disadvantages assembly programming has over high level language programming.

Performance is key to signal processing application because it directly translates into end-user features. A 10% lower clock speed may, for example, result in a 20% longer battery life. If the video decoding takes 80% of the CPU-cycle budget instead of 90%, there are twice as many cycles available for audio processing. This coupling of performance to end-user features is characteristic of many of the real-time applications in which DSP processors are applied.

The fixed-point and named address space extensions of *Embedded C* provide the programmer with direct access of these features in the target processor, thus significantly improving the performance of applications. The hardware I/O extension is a portability

[1] Van Eeghenstraat 100, 1071 GL Amsterdam, The Netherlands, Phone (+31)206646416, http://www.ace.nl.

feature of *Embedded C*. Its goal is to allow easy porting of device driver code between systems.

The focus of this paper is on the performance improving features of *Embedded C*.

1.1. Typical DSP Architectures

A look into the typical architecture of DSP processors is required to understand the need for an extension to C. DSP processors have a highly specialized architecture to achieve the performance requirements for signal processing applications within the limits of cost and power consumption set for consumer applications.

Unlike a conventional Load-Store (RISC) architecture, DSP processors have a data path with memory access units that directly feed into the arithmetic units. Address registers are taken out of the general purpose register file and placed next to the memory units in a separate register file.

A further specialization of the data path is the coupling of multiplication and addition to form a single cycle Multiply-ACcumulate unit (*MAC*). It is combined with special purpose accumulator registers, which are separate from the general purpose registers.

Data memory is segmented and placed close to the MAC in order to achieve the high bandwidths that are required to keep up with the streamlined data path. Limits are often placed on the extend of memory addressing operations.

The localization of resources in the data path saves many data movements that typically take place in a Load-Store architecture.

The most important common arithmetic extension to DSP architectures is the handling of saturated fixed-point operations by the arithmetic unit. Fixed-point arithmetic can be implemented with little additional cost over integer arithmetic. Automatic saturation (or clipping) significantly reduces the number of control flow instructions needed for checking overflow explicitly in the program.

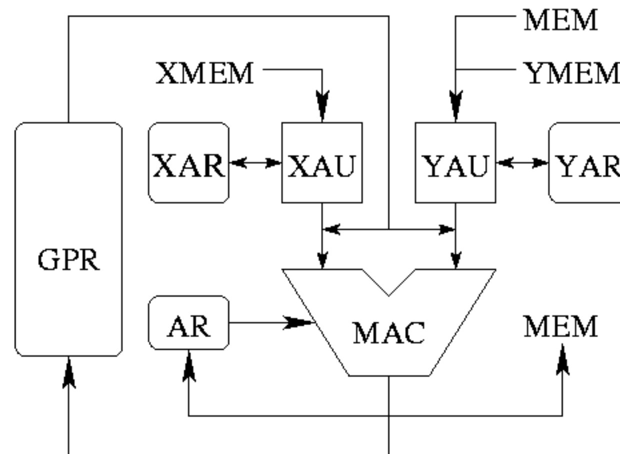

Figure 1*: Example DSP processor architecture with dual input memory data path and MAC unit.*

Figure 1 shows a picture of a typical DSP architecture. It shows the extended data path giving direct access to X and Y memory using X and Y addressing units (X/YAU). The addressing units have their own address registers (X/YAR) to implement address

post increment operations without having to access the general purpose registers (GPR). The addressing units are further specialized to implement circular buffer access, which is useful for implementing sliding windows over signal data.

1.2. Changing Requirements and the Role of the Compiler

DSP architectures are not easy to program optimally, either by hand or with a compiler. Manual assembly programming is awkward because of the non-orthogonality of the architecture and arbitrary restrictions that can be in place. Modern compilers can deal with non-orthogonality reasonably well, but are not good at exploiting the special features that DSP processors have in place.

The current state of the art embedded applications, mobile phones for example, are implemented using two processors. One processor is a low-power RISC processor that takes care of all control processing, user interaction and display management. It is programmed in a high level language making use of a software development kit that includes a compiler. The other processor is a DSP, which takes care of all of the signal processing. The signal processing algorithms are typically hand coded in assembly.

Changes in technological and economic requirements make it more and more expensive to continue programming the DSP processor in assembly. Staying with the mobile phone as an example, the signal processing algorithms required become increasingly complex. Features such as stronger error correction and encryption must be added. Communication protocols become more sophisticated and require much more code to implement. In certain market areas, multiple protocol stacks are implemented to be compatible with multiple service providers. In addition, backward compatibility with older protocols is needed to stay synchronized with provider networks that are in a slow process of upgrading.

On the economic side, time to market for new technology puts increasing pressure on design time. In 2004, the number of mobile phones that is expected to be sold world-wide is in the order of 500 million [2]. In the western world, the time to replacement for mobile phones is between 1 and 2 years, and is driven by new features and fashion. To stay ahead in this market requires extremely fast and streamlined design projects. Assembly programming has no place in this world. Assembly programming is error prone and slow. Assembly programs are difficult to maintain and make a company dependent on a few specialists. By definition, assembly programs are non-portable. Legacy code makes it extremely expensive to switch to a new technology. These dependencies make a company vulnerable to changes of its employees and its supplier chain.

1.3. The Programming Mismatch

Today, most embedded processors are offered with a C compiler. Despite this, programming DSP processors is still done in assembly for the signal processing parts or, at best, by using assembly written libraries supplied by the manufacturer. The key reason for this is that although the architecture is well matched to the requirements of the signal processing application, there is no way to express the algorithms efficiently and in a natural way in standard C. Saturated arithmetic, for example, is required in many algorithms and supplied as a primitive in many DSP processors. However, there is no such primitive in standard C. To express saturated arithmetic in C requires comparisons,

conditional statements and correcting assignments. Instead of using a primitive, the operation is spread over a number of statements that are difficult to recognize as a single primitive by a compiler.

1.4. Enter Embedded C

Embedded C is designed to bridge the performance mismatch between the signal processing algorithms, standard C and the architecture. It extends the C language with the primitives that are needed by signal processing applications and that are commonly provided by DSP processors. The design of *Embedded C* is based on *DSP-C*. *DSP-C* [1] is an industry designed extension of C with which experience was gained since 1998 by various DSP processor manufacturers in their compilers. For the development of *DSP-C*, by ACE, cooperation was sought with embedded application designers and DSP processor manufacturers. The *Embedded C* extension, like *DSP-C*, is designed in the spirit of the C language, applying the same rules to the *Embedded C* primitives that also hold for other primitives in C.

Embedded C makes life easier for the application programmer. The primitives provided by *Embedded C* are the primitives that fit the conceptual model of the application. The *Embedded C* extension to C unlocks the high performance features of DSP processors for C programmers. Assembly programming is no longer required for a vast body of performance critical code. Maintainability and portability of code are the key winners in this process.

2. Embedded C Features

The features introduced by *Embedded C* are fixed-point and saturated arithmetic, segmented memory spaces and hardware I/O addressing. The description that follows describe these features from the language design perspective, as opposed to the programmer's or processor architecture perspective. For the details and language definition of *Embedded C*, see [3].

2.1. Arithmetic

Embedded C adds two new primitive types `_Fract` and `_Accum`, and one type qualifier named `_Sat`. The underscores are included in these new keywords to ensure compatibility with existing applications. Typically, an implementation provides the more convenient macros `fract`, `accum` and `sat` in the include file `<stdfix.h>`.

2.1.1. The _Fract Type

The `_Fract` type offers fixed-point data types that have a value range of `[-1.0, +1.0>` (**-1.0** included but not **+1.0**). This is conveniently implemented using the two-complement arithmetic typically used for integer arithmetic. In two-complement notation, the dot of the fixed-point value is imagined right after the sign bit, before all value bits. The first value bit represents **0.5**, the second **0.25**, etc. A fixed-point number has no integer part.

The *Embedded C* language does not specify the exact accuracy of the fixed-point types although a minimum accuracy is defined to which an implementation must comply. The `_Fract` type can be qualified with the existing qualifiers `short` and `long` to

define three different fixed-point types. The range of these types is the same, **`[-1.0, +1.0>`**, but the accuracy should be equal or get better when moving from **`short _Fract`** to **`_Fract`** to **`long _Fract`**.

2.1.2. The `_Accum` Type

The **`_Accum`** is also a fixed-point type and can also be qualified with **`short`** and **`long`**. The three resulting **`_Accum`** types must match the three **`_Fract`** types in terms of accuracy, the number of bits in the fraction. Additionally, the **`_Accum`** types have an integer part in their value. So, the range of an **`_Accum`** value may be **`[-256.0, +256.0>`**. Again, the number of integer bits is not specified in the *Embedded C* definition.

The accumulator type matches the accumulator registers of typical DSP processors. The aim of these registers is to keep intermediate arithmetic results without having to worry about overflow.

2.1.3. The `_Sat` Qualifier

The **`_Sat`** qualifier can be applied to fixed-point types. It makes that all operations with operands of **`_Sat`** qualified type are saturated. It does not change the storage representation. Saturation means that if overflow occurs in an operation, the result will be set to the upper bound or lower bound of the type. For example, computing **`-0.75 + -0.75`** results in **`-1.0`** under saturated fixed-point arithmetic.

Saturated arithmetic is important for signal processing applications because they often operate close to the boundaries of the arithmetic domain in order to get the best signal to noise ratio. This is unlike integer processing in C, which is usually considered "large enough" and needs bound checks only at specific places.

2.1.4. The `unsigned` Qualifier

The **`unsigned`** qualifier (already existing in standard C) can also be applied to the fixed-point types, providing arithmetic domains starting from **`0.0`**. The range of the **`_Fract`** type becomes **`[0.0, 1.0>.`**

Unsigned arithmetic is typically used in image processing applications, but it is not universally present on all DSP processors.

2.1.5. Arithmetic Operations

The arithmetic operations for **`_Fract`** and **`_Accum`** include all those defined for the **`int`** type, but exclude **`~`**, **`&`**, **`|`** and **`^`**.

2.1.6. Conversions

Within the fixed-point hierarchy, the usual implicit conversions are defined. For example, the promotion of **`_Fract`** to **`unsigned _Fract`** or **`long _Fract`** is automatic; **`unsigned _Fract`** can be promoted to **`_Accum`**, with similar promotions for the **`long`** qualified variants.

Implicit conversions between fixed-point types and other types are fully defined. It is possible to write mixed type expressions. Conversions in mixed expressions are based on the rank order, which is **`int`**, **`_Fract`**, **`_Accum`**, **`float`**. Some extensions were made

to the usual handling of mixed arithmetic, in particular to make an expression like **`3 * 0.1r`** (where **`r`** denotes a fixed-point constant) meaningful. Under the usual arithmetic rules, the value **3** has to be converted to a fixed-point value first which is out of range and would lead to a meaningless result. With the extended rules, the intended outcome of **`0.3r`** is obtained.

2.1.7. Fixed-Point Design Rationale

An alternative to the current choice in the fixed-point design is to allow the programmer to specify exactly the number of relevant bits of the fixed-point types, or even to allow the programmer to specify the number of bits for every fixed-point variable. In this way, the implementation could guarantee the outcome of the computations.

Such a design would raise the abstraction level of the *Embedded C* language and increase the portability of code. However, it would also completely bypass the rationale of *Embedded C*, which is to provide a good match between the language and the performance increasing features of the processor. Enforcing an implementation of *Embedded C* to implement, for example, a 40 bit **`_Accum`** type on a processor that offers only 24 bit accumulators, is extremely awkward and would be highly inefficient. In that case *Embedded C* would be unusable for its purpose, which is to provide the programmer with access to the high performance features of the processor.

2.2. Multiple Address Spaces

Embedded C uses address space qualifiers to identify specific memory spaces in variable declarations. There are no predefined keywords for this, as the actual memory segmentation is left to the implementation. As an example, assume that **`X`** and **`Y`** are memory qualifiers. The definition:

```
X int a[25] ;
```

means that **`a`** is an array of 25 integers which is located in the **`X`** memory. Similarly (but less common):

```
X int * Y p ;
```

means that the pointer **`p`** is stored in the **`Y`** memory. This pointer points to integer data that is located in the **`X`** memory.

If no memory qualifiers are used, the data is stored into unqualified memory.

For proper integration with the C language a memory structure is specified, where the unqualified memory encompasses all other memories. All unqualified pointers are pointers into this unqualified memory. The unqualified memory abstraction is needed to keep the compatibility of the **`void *`** type, the **`NULL`** pointer and to avoid duplication of all library code that accesses memory through pointers that are passed as parameters.

2.3. I/O Hardware Addressing

The motivation to include primitives for I/O hardware addressing in *Embedded C* is to improve the portability of device driver code. In principle, a hardware device driver

should only be concerned with the device itself. The driver operates on the device through device registers, which are device specific.

However, the method to access these registers can be very different on different systems, even though it is the same device that is connected. The I/O hardware access primitives aim to create a layer that abstracts the system specific access method from the device that is accessed. The ultimate goal is to allow source code portability of device drivers between different systems.

In the design of the I/O hardware addressing interface, three requirements needed to be fulfilled:

- The device drive source code must be portable.
- The interface must not prevent implementations to produce machine code that is as efficient as other methods.
- The design should permit encapsulation of the system dependent access method.

The design is based on a small collection of functions that are specified in the **`<iohw.h>`** include file. These interfaces are divided into two groups, one group providing access to the device, the second group is for maintaining the access method abstraction itself.

2.3.1. Accessing the Device

To access the device, the following functions are defined by *Embedded C*:

```
unsigned int iord(  ioreg_designator );
void iowr(  ioreg_designator, unsigned int value );
void ioor(  ioreg_designator, unsigned int value );
void ioand( ioreg_designator, unsigned int value );
void ioxor( ioreg_designator, unsigned int value );
```

These interfaces provide read and write access to device registers, as well as typical methods for setting and resetting individual bits. Variants of these functions are defined (with **buf** appended to the names) to access arrays of registers. Variants are also defined (with **`l`** appended) to operate with **long** values.

All of these interfaces take an I/O register designator **`ioreg_designator`** as one of the arguments. These registers designators are an abstraction of the real registers provided by the system implementation and hide the access method from the driver source code.

2.3.2. Managing I/O Register Designators

Three functions are defined for managing the I/O register designators. Although these are abstract entities for the device driver, the driver does have the obligation to initialize and release the access methods. Note that these functions do not access, or initialize, the device itself since that is the task of the driver. They allow, for example, the operating system to provide a memory mapping of the device in the user address space.

```
void iogroup_acquire( iogrp_designator );
void iogroup_release( iogrp_designator );
void iogroup_map( iogrp_designator, iogrp_designator );
```

The **iogrp_designator** specifies a logical group of I/O register designators; typically this will be all the registers of one device. Like the I/O register designator, the I/O group designator is an identifier or macro that is provided by the system implementation.

The `map` variant allows cloning of an access method when one device driver is to be used to access multiple identical devices.

2.4. Portability of Embedded C

By design, a number of properties in *Embedded C* are left implementation defined. This implies that the portability of *Embedded C* programs is not always guaranteed. *Embedded C* provides access to the performance features of DSP processors. As not all processors are equal, not all *Embedded C* implementations can be equal.

For example, suppose an application requires 24 bit fixed-point arithmetic and an *Embedded C* implementation provides only 16 bits because that is the native size of the processor. When the algorithm is expressed in *Embedded C*, it will not produce outputs of the right precision. In such a case, there is a mismatch between the requirements of the application and the capabilities of the processor. Under no circumstances, including the use of assembly, will the algorithm run efficiently on such a processor. *Embedded C* can not overcome such discrepancies.

Yet, *Embedded C* provides an great improvement in the portability and software engineering of signal processing applications. Despite many differences between specific DSP processors, there is a remarkable similarity in the special purpose features that they provide to speed up signal processing applications.

The contribution of *Embedded C* is that it standardizes the notation to access these features. By adding this to the C language, it is now feasible to write high-performance code for embedded and DSP processors in a high level language. The responsibility of the programmer to do the laborious resource planning tasks that assembly language programming requires is taken away.

2.5. C++ Compatibility

C++ compatibility was another topic of debate in the design of *Embedded C*. Preferably the extension should be expressed in such a way that it can be implemented as C++ classes. It implies that the extension should not depend on the use of type qualifiers. This, however, is against the “spirit of C” and leads to a long list of types that result when all combinations of qualifiers are expanded. While this would be feasible for the fixed-point types, it would still not provide a solution for the named address space qualifiers.

Hence the current design that follows standard C practice. In the case code must be written that is to be accepted by both a C and C++ compiler, one needs to define macros such as `unsigned_long_accum`, which should then expand into the right pattern for the specific compiler. For named address spaces there is no similar solution yet because these do not commonly appear in C++ compilers.

2.6. Features That Did Not Make It Into Embedded C

The current specification of *Embedded C* is not the final station. *Embedded C* is defined to be a common playground for extensions to the C language required by typical application and system areas. Although there were more proposals for extensions in *Embedded C*, the committee decided to include only those that have shown a certain level of maturity.

2.6.1. Circular Buffers

A Circular Buffer is a memory addressing feature that implements wraparound access to arrays. Circular buffers are typically used to efficiently model sliding windows over streamed input data. Instead of shifting the array for every new data element in the window, the window simply wraps around at the end of the array. Hardware support for this operation avoids control flow operations that are normally required to implement such wraparound.

Although this feature was already present (successfully) in *DSP-C*, the committee found that there were too many different implementations of circular buffer support in DSP processors to be able to define a single unified specification in C that can map efficiently to all current implementations.

2.6.2. Fixed-Point Complex Data Types

Complex data types are defined in the current *ISO C* standard (also known as C9X). These are defined for floating point types. Although an extension to the fixed-point types is logical, this was not incorporated in the current report yet.

2.6.3. Binary Coded Decimal Types

BCD types were briefly considered in the discussions on *Embedded C*. The area of applications for these types is so diverse (also beyond embedded processing) that they were dropped. The current report deals with binary types only.

2.6.4. Modulo Wrapping

An alternative method of handling overflow named **`_Modwrap`** was considered until late in the design of *Embedded C*. It would provide an alternative qualifier for overflow rounding next to **`_Sat`**, which should round fixed-point values modulo 1 in case of overflow. It was dropped mainly because the committee did not want to clutter the type system with another qualifier.

3. Example

Figure 2 shows an example of a FIR filter in *Embedded C*. It includes the fixed-point and address space features of *Embedded C*.

The first line defines **`coeff`**, an array of **`_Fract`** values located in the **`X`** memory. The first value of the (incomplete) initialization shows the notation of a **`_Fract`** value constant, with the **`r`** appended. On line 5 there is an example of an **`_Accum`** constant value, which has a **`k`** appended.

```
 1    X _Fract coeff[N] = { 0.7r, ... } ;
 2
 3    _Fract fir( Y _Fract *inp ) {
 4        int i ;
 5        _Accum sum = 0.0k ;
 6        for( i = 0 ; i < N ; i++ ) {
 7            sum += coeff[i] * (_Accum)*inp++ ;
 8        }
 9        return (_Sat _Fract)sum ;
10    }
```

Figure 2: *Example FIR filter in Embedded C.*

Line 3 gives the **`fir`** function type declaration. The function returns a **`_Fract`**. The argument points to an array of input values of type **`_Fract`** that is located in **`Y`** memory.

Line 5 defines an accumulator variable. Lines 6 to 8 define a loop of **`N`** iterations. The body of the loop is a single multiply-accumulate statement.

In line 7 there is an explicit conversion of the input value to the **`_Accum`** type. This is to make sure that the processor's MAC unit can be used. Without the explicit conversion, the rules of arithmetic in C specify that the intermediate result of the multiplication must be a **`_Fract`** type; it does not allow the partial result to be kept with a higher accuracy. However, the practice of a typical MAC unit implementation is that the partial result *is* kept with a high precision. The explicit conversion makes the use of a MAC possible.

Also important in line 7 is loading of the coefficient value and the input value from two different memories, **`X`** and **`Y`**. This corresponds to the DSP processor's architecture and allows for maximum bandwidth.

Line 9 makes an explicit saturating conversion from the **`_Accum`** type. It makes sure that the returned value gets the maximum or minimum **`_Fract`** value in case of overflow.

3.1. Alternative in C

To write a program in plain C that implements the semantics of the example is certainly possible but would lead to an implementation that will not use the performance improving features of the DSP target processor. Plain C has no awareness of multiple memories. This notion cannot be expressed, hence the implementation will not use the dual input pipeline of the DSP architecture. Similarly, fixed-point operations and saturation need to be expressed in terms of shifts and conditional execution. Given the number of different ways in which this can be done, it is almost impossible for the compiler to find out how this is best translated to DSP specific features.

4. Implementation, Performance

Given the *Embedded C* version of the example, an optimizing compiler can compile the loop into a single instruction running under zero-overhead loop control. This is as

good as an assembly version of the FIR filter can be. Besides knowledge of *Embedded C*, the required compiler techniques are:

- Recognition of zero-overhead loops
- Memory disambiguation based on multiple memories
- Transforming array access to a walking pointer
- Recognition of the MAC pattern
- Scheduling post-increment with memory access

All of these capabilities are well within the reach of modern compilers.

4.1. Performance Results for Embedded C

To show the improvements that are possible due to the use of *Embedded C*, the following tables show results of the commercial *DSP-C* implementation for the NEC μPD77016 processor. This compiler was made with the ACE CoSy compiler development system.

To make the comparison, four applications were selected that were originally implemented in *DSP-C*. These applications were then re-implemented for comparison using plain *ISO C* only. A single *DSP-C* capable compiler was used in all comparisons. This is possible because a *DSP-C* compiler is also fully *ISO C* compliant.

Table 1 shows the improvements in code size when *DSP-C* is used. The first of the four applications is a piece of control code. It was included in the test to show the contrast between dealing with signal processing code and control code.

In control code there is typically little use for the *DSP-C* extensions. This is confirmed in the code size that is reduced by only 6% in the *DSP-C* variant. In control code there is usually no signal processing arithmetic done, but the multiple memory and circular array primitives are sometimes applicable.

Code Size	*ISO C*	*DSP-C*	Ratio
Control	442	414	1.1
SP1	350	90	3.9
SP2	2152	1155	1.9
SP3	3807	2781	1.4

Table 1: *Code size comparison between four applications written in both standard ISO C and DSP-C, both compiled with the same compiler. The first application is a control code application, the other three are signal processing applications. Measurements are given in bytes.*

For the three signal processing applications SP1, SP2 and SP3, the code savings are substantial. Application SP1 is even reduced to a quarter of its original size, but note that this was a small application to start with, containing only DSP specific code. For the other two, with more administrative code in them, the savings are still substantial.

These results show that *Embedded C* is not only important to achieve high performance, but additionally leads to significant code savings.

Table 2 shows the improvements in execution cycles for application run time. For the control code, the improvement is minimal. The 6% code size improvement did not

translate into a similar speed improvement, this is because the size improvements did not occur in one of the performance critical loops of the control code application.

Clock Cycles	*ISO C*	*DSP-C*	Ratio
Control	3946	3890	1.0
SP1	5144	550	9.4
SP2	168546	48064	3.5
SP3	2822	349	8.1

Table 2: *Execution time comparisons between the ISO C and* DSP-C *implementations of the four applications. The run time number of clock cycles is reported.*

The performance improvements for the signal processing applications are remarkable. They demonstrate the tremendous effect that the specific features of DSP processors can have when used to their full advantage. The key issue with DSP processors is that everything must fit just right before the maximum benefit can be obtained from the tightly designed processor data path. In the case of SP1, a 10 times performance improvement is measured. As the performance improvements outweigh the code size improvements, this shows that for the signal processing applications the actual hot spots of the programs are attacked by *DSP-C*.

Based on these figures and more extensive comparisons it is impossible to claim that *DSP-C* will always lead to a specific factor of performance improvement for signal processing applications. This factor depends on how well the application fits the processor architecture.

These figures are a proof of concept for *Embedded C*. They show that for real world applications *Embedded C* enables the use of the high speed extensions that are found in modern DSP processors. It is possibly to effectively program DSP processors in C!

5. History, Status and Future of Embedded C

Embedded C has its roots in *DSP-C*, which is an industry standard extension for C with features for DSP processing. Implementations of *DSP-C* exist for the Philips REAL, Adelante's Saturn DSP, NEC's µPD7701x family, TI's TMS320C54x, Analog Devices' SHARC and Siroyan's OneDSP. Other *DSP-C* supporters or users are ARC International, Atair software, DeSOC Technology, Japan Novel, Mentor Graphics, NullStone and Toshiba. Mentor Graphics, in particular, implements the XRAY debugger that allows source level debugging of *DSP-C* programs. Check the *DSP-C* web-site http://www.dsp-c.org for up-to-date information.

ACE submitted *DSP-C* for standardization by the ISO working group on C. This resulted in the ISO technical report on *Embedded C* [3]. Although *Embedded C* differs slightly from *DSP-C*, it fully subscribes to its design rationale. It bridges the gap between signal processing applications, plain C, and signal processing hardware.

It is expected that the current *DSP-C* implementations will also support *Embedded C* in the future because of the strong similarities between the two.

The first ballot on the draft *Embedded C* document was in 2002. Feedback from this ballot was incorporated into the 2003 draft. The final report was ratified in February 2004.

The definition of the current report does not exclude further improvements. On the contrary, the committee was relatively conservative in its choices in order to allow further experience to be gained with alternative solutions in the field. This may lead to future revision and extension of the report. Check out http://www.embedded-c.org to stay up to date.

6. Conclusion

Embedded C is a relatively small extension to the C language, but its impact on the programmability of embedded and DSP processors in particular is enormous. Specialized high performance features are the reasons why DSP processors exists. Without the *Embedded C* extension these features are inaccessible to the high level language application programmer.

Today it is still common to program these processors in assembly language. The industry cannot afford the increasing time to market that assembly programming of ever more complicated applications incurs. Moreover the inherit dependency on a specific processor and few highly specialized assembly programmers incurs great risk and paralyzes future developments.

Embedded C offers a practical solution with proven results. It is being adopted by more and more compiler developers. With the ratification of the ISO technical report, the *Embedded C* approach is the standard solution for high level language programming of the many billions of embedded processors out in the field.

References

[1] ACE. DSP-C, an extension to ISO/IEC IS 9899:1990. Technical Report CoSy-8025P-dsp-c, ACE Associated Compiler Experts bv, Amsterdam, The Netherlands, 1998. Downloadable from http://www.dsp-c.org.

[2] EETIMES UK. IDC predicts strong mobile phone growth in 2004. http://www.eetuk.com, 2003.

[3] JTC1/SC22/WG14. Extensions for the programming language C to support embedded processors. Technical report, ISO/IEC, 2003. Downloadable from http://www.dkuug.dk/JTC1/SC22/WG14.