WG14 N907

Proposals for Extensions for the programming language C to support embedded processors

Willem Wakker
ACE Consulting bv

Version 1.2, March 2000

Contents

1. Introduction

1.1 Why are extensions of the C language to support embedded processors necessary?

In the fast growing market of embedded systems there is an increasing need to write application programs in a high-level language such as C. Basically there are two reasons for this trend: programs for embedded systems get more complex (and hence are difficult to maintain in assembly language) and the different types of embedded systems processors have a decreasing lifespan (which implies more frequent re-adapting of the applications to the new instruction set). The code re-usability achieved by C-level programming is considered to be a major step forward in addressing these issues.

Various technical areas have been identified where functionality offered by processors (such as DSPs) that are used in embedded systems cannot easily be exploited by applications written in C. Examples are fixed-point operations, usage of different memory spaces, low level I/O operations and others. The current proposal addresses only a few of these technical areas.

Embedded processors are often used to analyse analogue signals and process these signals by applying filtering algorithms to the data received. Typical applications can be found in all wireless devices. The common datatype used in filtering algorithms is the fixed point datatype, and in order to achieve the necessary speed, the embedded processors are often equipped with special hardware to support that datatype. Standard C does not provide support the fixed point arithmetic operations, currently leaving programmers with no option but to hand-craft most of their algorithms in assembler. To overcome this problem, it is proposed to add a fixed point datatype to the language, definable in a range of precision and saturation options. In this manner, fixed point data is supported as easily as integer and floating point data throughout the compiler, including the critical optimisers leading to highly efficient code.

Typical for the mentioned filtering algorithms is the usage of polynomials whereby data from one source (inputvalues) is multiplied by coefficients coming from another source (memory). Ensuring the simultaneous flow of data and coefficient data to the multiplier/accumulator of processors designed for FIR filtering, for example, is critical to their operation. In order to allow the programmer to declare the memory space from which a specific data object must be fetched, this proposal suggests to support the multiple memory spaces of dual Harvard architectures DSP processors. As a result, optimising compilers can utilise the ability of these processors to read data from two separate memories in a single cycle to maximise execution speed.

Another feature of many DSP algorithms, including FIR filters and FFTs, is that they frequently use the same block of data over and over again, working from the start of the block to the end and then looping back to the beginning. Although the optimum way of achieving this is to create a circular buffer, such buffers are not easily defined in standard C programs. This proposal therefore includes a language extension that allows the size and memory space of circular buffers to be defined so that they can be easily generated during program compilation.

1.2 About this document

This document is written to provide a basis for discussion in WG14 on the development of the NP on C extensions for embedded processor support. It is intended to provide the background and rationale for choices made, as well as for the possibilities rejected. It should also serve as the basis for the future Rationale document that is to be produced under the same NP. Although (obviously) proposals will be presented to augment the C language, the actual definitions of new syntax and semantics will be specified in the Technical Report to be produced as main topic of the NP.

The main part of this (version of this) document will be the discussion of the background of fixed-point datatypes, and the ways in which these datatypes can be incorporated in the C typesystem. Both the C aspects (type specifiers, type conversions, constant definitions and the relationship with the standard C datatypes) as well as typical fixed-point arithmetic related issues will be addressed. Where possible, issues that arise from the fixed point arithmetic (like saturation) will be formulated in a general way so that they can be applied to non fixed-point arithmetic as well. Although embedded processors typically work with binary (radix=2) fixed-point datatypes, the inclusion of decimal (radix=10) fixed-point datatypes or the inclusion of a generalized fixed-point datatype (like the Scaled datatype as defined in the Language-independent datatypes standard) will be considered. Prior art (for instance as described in WG14 N854) will be considered while developing the specifications.

Other technical areas to be addressed (depending on input received) are different memory spaces and the handling of circular buffers.

1.3 Preciseness/level of the specification

The following is a list of design criteria to be used in developing this specification:
- the specification should be low-level enough (be specific about bits, sizes, spaces etc) to allow compilers to fully utilize the hardware capabilities of the embedded processors that support in hardware the described functionality;
- the specification should be high-level enough (be specific about arithmetic properties, rounding modes, relationship with other language elements) to allow programmers to write portable algorithms;
- the specification should be concise enough to allow many compilers implementers to implement the specifications, and yet be useful for the programmers of the specific algorithms;
- it should be possible for an implementation to make use of the underlying hardware wherever possible and implement the non-supported features differently (emulation libraries? using floating point hardware?) without hampering the portability of the program.

1.4 Discussion of various approaches to language extension

Various approaches to include/support the described new functionality (but specifically fixed point arithmetic) are possible: the definition of new language elements (as done in this proposal), definition of a set of intrinsic library routines, or a C++ class approach. Each approach has pro's and con's. It is however true for all approaches that, if it is the intention to make optimal use of the hardware support of the underlying processor, extensive additions to an existing compiler should be prepared to achieve this goal. In other words: also for specifications (like intrinsics and class libraries) that can be seen as building on existing standards and for which it is not really necessary to change an existing compiler, if optimal code is required then the compiler should 'know' about the underlying machine instruction set. As a consequence, the chosen method is largely independent of the amount of work that needs to be done in adapting a compiler to support all required features.

The pro's and con's can be summarized as follows:

	Pro	Con
Language extensions	- natural (though extensive) addition to the type system	- nothing can be done without an adapted compiler
Intrinsics, integer datatypes used in stead of fixed point datatypes	- easy to include as a stand-alone add-on package	- everything is an integer datatype, and it is left to the user to select the right parameters with the right intrinsic functions: error prone - fixed point constants obscured as integer constants
C++ class approach	- intuitive language extension; burden on the class writer rather than on the compiler writer - easy to experiment with, easy for modeling purposes	- for embedded environments space is of the utmost importance (a slightly larger memory means a few extra cents per unit, this for large volume production implies huge increase in costs). Even when the compiler is optimised for the target instructions, the other runtime overhead (in size) makes C++ less suitable - in comparison with C, C++ is a less easy to use language for programmers that were used to program only in assembly code

It is the intention to define the extensions in such a way that the resulting document can be used both in a C context and in a C++ context. This implies that care should be taken to define the extensions in the 'intersection' of the C standard and C++ standard, thereby avoiding to make use of those C features that are not defined in the C++ standard. However, if it is natural or convenient to make use of a new C feature, somewhere (possibly in an Annex) it should be defined what the consequences are if that feature is not available.

2. Fixed point types

2.1 Fixed point types and values: an intuitive approach

The set of representable floating point values (which is a subset of the real values) is characterized by a sign, a precision and the position of the radix point. For those values that are commonly denoted as *floating point* values, the characterizing parameters are defined within a format (such as the IEEE formats or the VAX floating point formats), usually supported by hardware instructions, that defines the size of the container, the size (and position within the container) of the exponent, and the size (and position within the container) of the sign. The remaining part of the container then contains the mantissa. [The formats discussed in this section are assumed to be binary floating point formats, with sizes expressed in bits. A generalization to other radixes (like radix-10) is possible, but not done here.] The value of the exponent then defines the position of the radix point.
Common hardware support for floating point operations implements a limited number of floating point formats, usually characterized by the size of the container (32-bits, 64-bits etc); within the container the number of bits allocated for the exponent (and thus for the mantissa) is fixed. For programming languages this leads to a small number of distinct floating point datatypes (for C these are **float**, **double**, and **long double**), each with its own set of representable values.

For *fixed point* types, the story is slightly more complicated: a fixed point value is characterized by its precision (the number of databits in the fixed point value) and an optional signbit, while the position of the radix point is defined implicitly (i.e., outside the format representation): it is not possible to deduct the position of the radix point within a fixed point datavalue (and hence the value of that fixed point datavalue!) by simply looking at the representation of that datavalue. It is however clear that, for proper interpretation of the values, the hardware (or software) implementing the operations on the fixed point values should know where the radix point is positioned. From a theoretical point of view this leads (for each number of databits in a fixed point datatype) to an infinite number of different fixed point datatypes (the radix point can be located anywhere before, in or after the bits comprising the value).
There is no (known) hardware available that can implement all possible fixed point datatypes, and, unfortunately, each hardware manufacturer has made its own selection, depending on the field of application of the processor implementing the fixed point datatype. Unless a complete dynamic or a parameterized typesystem is used (not part of the current C standard, hence not proposed here), for programming language support of fixed point datatypes a number of choices need to be made to limit the number of allowable (and/or supported or to be supported) fixed point datatypes. In order to give some guidance for those choices, some aspects of fixed point datavalues and their uses are investigated here.

For the sake of this discussion, a fixed point datavalue is assumed to consist of a number of databits, one of which is the signbit. On some systems, the signbit can be used as an extra databit, thereby creating an unsigned fixed point datatype with a larger (positive) maximum value.
Note that the size of (the number of bits used for) a fixed point datavalue does not necessarily

equal the size of the container in which the fixed point datavalue is contained (or through which the fixed point datavalue is addressed): there may be gaps here!

As stated before, it is necessary, when using a fixed point datavalue, to know the place of the radix point. There are several possibilities.

- The radix point is located immediately to the right of the rightmost (least significant) bit of the databits. This is a form of the ordinary integer datatype, and does not (for this discussion) form part of the fixed point datatypes.
- The radix point is located further to the right of the rightmost (least significant) bit of the databits. This is a form of an integer datatype (for large, but not very precise integer values) that is normally not supported by (fixed point) hardware. In this section, these fixed point datatypes will not be taken into account.
- The radix point is located to the left of (but not adjacent to) the leftmost (most significant) bit of the databits. It is not clear whether this category should be taken into account: when the radix point is only a few bits away, it could be more 'natural' to use a datatype with more bits; in any case this datatype can easily (??) be simulated by using appropriate normalize (shift left/right) operations. There is no known fixed point hardware that supports this datatype.
- The radix point is located immediately to the left of the leftmost (most significant) bit of the databits. This datatype has values (for signed datatypes) in the interval [-1,+1), or (for unsigned datatypes) in the interval [0,1). This is a very common, hardware supported, fixed point datatype. In the rest of this section, this fixed point datatype will be called the *type-A* fixed point datatype. Note that for each number of databits, there are one (signed) or two (signed and unsigned) possible type-A fixed point datatypes.
- The radix point is located somewhere between the leftmost and the rightmost bit of the databits. The datavalues for this fixed point datatype (*type-B* fixed point datatypes) have an integral part and a fractional part. Some of these fixed point datatypes are regularly supported by hardware. For each number of databits N, there are (N-2) (signed) or (2*N-3) (signed and unsigned) possible type-B fixed point datatypes.
  Note: it is not strictly necessary that the signbit of a signed fixed point datavalue corresponds to the most significant bit of the of the unsigned fixed point datavalue with the same number of databits.

Apart from the position of the radix point, there are three more aspects that influence the amount of possible fixed point datatypes: the presence of a signbit, the number of databits comprising the fixed point datavalues and the size of the container in which the fixed point datavalues are stored.
In the embedded processor world, support for unsigned fixed point datatypes is rare; normally only signed fixed point datatypes are supported. However, to disallow unsigned fixed point arithmetic from programming languages (in general, and from C in particular) based on this observation, seems overly restrictive.

The concept *container* is used to identify the address and the size (expressed in bytes) of a fixed point datavalue. Since fixed point hardware support is sometimes built for a special purpose, requiring a certain precision, it is not necessarily the case that the number of databits in a fixed point datavalue can be expressed as a multiple of the number of bits per byte. In other words: it can very well be the case that on an 8-bit byte machine, a 10-bit unsigned

fixed point datatype is supported.
Some assumptions:
- the size of the container in bits is a multiple of the number of bits per byte for the machine, and the address of the container is a regular byte address;
- the number of databits in a fixed point datavalue is not greater than the number of bits in its container;
- the (machine) address of a fixed point datavalue is the (machine) address of (exactly) one of the bytes that form the container;
- if the size of the container in bits is greater than the number of bits needed for the fixed point datavalue, the remaining bits (called *paddingbits*) cannot be used for other purposes (it is not proposed to introduce packed fixed point datatypes; tricks like creating a union of a fixed point datatype and a structure containing bitfields are discarded); [Question: should implementations (if any) be supported that have say a 4-bit fixed point type (always left aligned in an 8-bit container) and a 12-bit fixed point type (always right aligned in a 16-bit container). In such an implementation two distinct fixed point datavalues could have the same address.]
- if there are paddingbits involved, it is assumed that still the (machine) address of (one of the bytes of) the container fully identifies the (machine) address of the fixed point datavalue; in other words: the alignment of the fixed point datavalue within the container is implicitly known (from its fixed point datatype designation);
- at programming level (i.e., in the programming language) all fixed point datatypes with the same valuespace (the same number of databits, same signedness, same position of the radix point) are the same; there is no distinction between these datatypes with respect to different alignment/padding strategies.

There are two further design criteria that should be considered when defining the nature of the fixed point datatypes:
- it should be possible to generate optimal fixed point code for various processors, supporting different sized fixed point datatypes (examples could include an 8-bit fixed point datatype, but also a 6-bit fixed point datatype in an 8-bit container, or a 12-bit fixed point datatype in a 16-bit container);
- it should be possible to write fixed point algorithms that are independent of the actual fixed point hardware support. This implies that a programmer (or a running program) should have access to all parameters that define the behaviour of the underlying hardware (in other words: even if these parameters are implementation defined).

With the above observations in mind, the following recommendations can be made.
1. Introduce signed and unsigned fixed point datatypes, and use the existing **signed** and **unsigned** keywords (in the 'normal' C-fashion) to distinguish these types. Omission of either keywords implies a signed fixed point datatype.
2. Introduce a new keyword and *type-specifier* **fixed** (similar to the existing keyword **int**), and define the following five *standard signed fixed point types*: **char fixed**, **short fixed**, **fixed**, **long fixed** and **long long fixed**. The supported (or required) underlying fixed point datatypes are mapped on the above in an implementation-defined manner, but in a non-decreasing order with respect to the number of databits in the corresponding fixed point datavalue. Note that there is not necessarily a correspondence between a fixed point datatype designator and the type of its container: when an 18-bit and a 30-bit fixed point

datatype are supported, the 18-bit will probably have the **short fixed** type and the 30-bit type will probably have the **fixed** type, while the containers of these types will be the same.

An open question is: should a distinction be made between type-A (no integral part, only fractional part) and type-B (integral part and fractional part) fixed point datatypes by introducing one or more additional keywords.
Arguments to have extra keywords are:
- the type-A and type-B fixed point datatypes are (probably) used for different purposes, hence it might be useful to be able to make a distinction between them at programming level;
- the **char fixed** datatype looks awkward, and should possibly not be allowed; similarly, it is not likely that the number of databits in a fixed point datatype will ever be in the range of other **long long** datatypes, so also the corresponding **long long fixed** datatype might not be recommended. This leaves us with only three 'regular' fixed point datatypes, which might not be enough. The introduction of a separate class of fixed point datatypes may solve this problem.

Arguments against extra keywords are:
- why make a distinction between two conceptually identical datatypes? However note that this has been done before in C: **short** - **int** - **long** and **float** - **double**;
- the introduction of yet another keyword.

Assuming that it is decided to go for one or more additional keywords, here are some points to consider.
- From a conceptual point of view, both type-A and type-B fixed point datatypes form part of the **fixed** datatype, as introduced above. Therefore, an obvious choice would be to introduce two more keywords to qualify **fixed** to get to something like **typeA fixed** and **typeB fixed**. Better choices of words would be **fract fixed** (for type-A fixed point datatypes) and **accum fixed** (for type-B fixed point datatypes; **accum** indicates where type-B fixed point datatypes are usually used for: to accumulate (type-A) fixed point datavalues).
- If this is considered to be too verbose, then it can be argued that, since the **fract fixed** datatype is the most common one, **fixed** could be considered to be a meaningful abbreviation of **fract fixed**, and then **accum fixed** could be abbreviated to **accum**. Whether the full form and the abbreviated form are allowed is something to be discussed.

Further recommendations:

3. When more fixed point datatypes are needed, or when it is considered necessary to give a programmer access to more precise fixed point datatype specification, an approach similar to the **<stdint.h>** approach could be taken, whereby (as an example) **fixed_le*N*_t** could designate a fixed point datatype with at least N databits, while **accum_le*M*_le*N***_t could designate a fixed point datatype with at least *M* integral bits and *N* fractional bits.
   Defining fixed point datatypes in this general fashion offers the possibility to describe in a more precise manner the properties of the fixed point datatypes, and the relationships with other datatypes. But, on the other hand it creates a (possible) false sense of portability: if not every system supports the general approach (and certainly the current hardware does not do this) then either complete software emulation should be supported, or shortcuts

(i.e., mapping to other fixed point or floating point datatypes) should be implemented.

4. In order for the programmer to be able to write portable algorithms using fixed point datatypes, information on (and/or control over) the nature and precision of the underlying fixed point datatypes should be provided. The normal C-way of doing this is by defining macro names (like **SHORT_FIXED_FRAC_BITS** etc.) that should be defined in an implementation-defined manner.

2.2 Fixed point datatypes, based on LID (ISO/IEC 11404)

ISO/IEC 11404 - Language-Independent Datatypes (LID) describes the Scaled datatype as a family of datatypes whose value spaces are subsets of the rational value space (the Rational datatype being defined as a pair of unbounded integers with the obvious meaning), each individual (scaled) datatype having a fixed denominator defined by two parameters (*radix* and *factor*) whereby the value of the denominator is defined by *radix* raised to the power *factor*.

When the LID definition is used as a basis for fixed point datatypes in C, a relation between the parameters defining the LID Scaled datatype and the corresponding C fixed point datatype needs to be established. The parameters are:

1. The value of the *radix* parameter.
   Although in theory any integer value larger than one could be used as value for the *radix* parameter, only the values 2 and 10 are (currently) supported by hardware implementations (as binary fixed point values and BCD values).
   Since embedded processors normally only support binary fixed point, this specification only discusses fixed point datatypes with radix == 2, but the general requirements and characteristics are also valid for other radix values.

2. The value of the *factor* parameter.
   In principle any integer value (positive or negative) can be used as value for the *factor* parameter: this value specifies the scaling factor (the number of places by which the databits are to be shifted). For the type-A fixed point datatypes, the value of the *factor* parameter equals the number of databits minus one (for signed fixed point datatypes) in the fixed point datavalue; for the type-B fixed point datatypes the value of the *factor* parameter is greater than zero, but less than the number of databits minus one in the (signed) fixed point datavalue.

3. The size of the numerator.
   In LID, integers are (in principle) unbounded; in C various integer types are supported with varying sizes. However, the size (in bits) of the numerator establishes the (maximum) precision of a fixed point datatype. Therefore, although the size of the numerator is (from an LID point of view) not really a parameter defining the Scaled datatype, for the correspondence between the Scaled datatype and the C fixed point datatypes it is necessary to take the size of the numerator into account.

When omitting the *radix* parameter this leads to signed and unsigned **LID_fixed_le*R*_*F*_t** fixed point datatypes, whereby the size of the numerator (i.e. the number of databits in the fixed point datavalue) is at least *R*, and the value of the *factor* parameter is indicated by *F*.

Some observations on the relationship between the thus defined fixed point datatypes and the fixed point datatypes defined in the previous section (assuming that the recommendations are followed):

- With the **char**, **short** and **long** keywords as 'size indicators' for the various fixed point datatypes, it is not the case that there is a one-to-one correspondence between the size of the size indicator and *R*.
- The **fixed_le***M***_le***N***_t** corresponds to the **LID_fixed_le***R***_***F***_t** type when (*M*+*N*)==*R* and *N*==*F*.
- The **LID_fixed_le***R***_***F***_t** notation can easily be used for fixed point datatypes where the radix point is located outside the databits; the **fixed_le***M***_le***N***_t** notation cannot be used to describe those datatypes (assuming that the required precision is much smaller than size of the container).

2.3 Saturation

Saturation is a mode associated with a variable. When a value is assigned to a saturated variable and that value is too large (or too small) to be represented by the variable, the maximal (or minimal) value (according to the type of the variable) is assigned instead. In other circumstances an overflow condition would have been created.

In order to implement saturation it is proposed to introduce a new keyword **sat** that can be used as qualifier in type specifications.

Note that with the above definition, the effect of saturation is directly related to the number of databits in the datatype. A more general definition could introduce a limit on the values of a variable, while this limit is not the maximal value that can be represented (think of a 4-bits unsigned datatype that only is supposed to hold the values 0-9). With such a definition, saturation becomes a special case of limiting values. Since such a general scheme is not supported in hardware (or used in software) it is not proposed (or discussed) here. Still, this generalised approach may be taken into account when defining the name of the keyword (**lim** or **limited** instead of **sat**?).

With the above definition, saturation is only established during an assignment. When saturation is also needed for intermediate results in expressions, there are several ways to accomplish this:

- by inserting type casts in the expression;
- similar to the **STDC FP_CONTRACT** pragma, by defining a pragma that requires that all operations in an expression yield a saturated result;
- by including saturation in the usual arithmetic conversions so that any operation involving a saturated operand yields a saturated result;
- by defining specialized operators for each operator (including the assignment operator) that might yield a saturated result.

The above possibilities are not mutual exclusive; a mix is possible.

Note that, although saturation is usually connected with fixed point arithmetic, the usage of the saturation keyword is not necessarily limited to fixed point arithmetic.

Attention:

- care should be taken in those cases whereby paddingbits are present directly to the left of the most significant bits of the integral part of a saturated variable;
- some hardware has fixed point hardware registers, whereby a copy to memory automatically implies saturation.

2.4 Usual arithmetic conversions, type casts

It is proposed to situate the fixed point datatypes 'between' the integer datatypes and the floating point datatypes: if only integer datatypes are involved then the current standard rules (cf. 6.3.1.1 and 6.3.1.8) are followed, when fixed point operands but no floating point operands are involved the operation will be done using fixed point datatypes, otherwise everything will be converted to the appropriate floating point datatype.

Since it is likely that an implementation will support more than one (type-A and/or type-B) fixed point datatype, in order to assure arithmetic consistency it should be well-defined to which fixed point datatype a type is converted to before an operation involving fixed point and integer datavalues is performed. There are several approaches that could be followed here:

- define that the result of any operation on fixed point datatypes should be as if the operation is done using infinite precision. This gives an implementation the possibility to choose an implementation dependent optimal way of calculating the result (depending on the required precision of the expression by selecting certain fixed point operations, or, maybe, emulate the fixed point expression in a floating point unit), as long as the required result is obtained.
- define an (implementation defined?) 'extended' fixed point datatype to which every operand is converted before the operation. It is then important that the programmer has access to the parameters of this extended fixed point type in order to control the arithmetic and its results. As an example, this extended fixed point datatype could either be the 'largest' type-B fixed point datatype (if supported), or the 'largest' type-A fixed point datatype.
- define the 'usual' C litany of possibilities ("if one of the operands has type …"). The conversion strategy then becomes a crucial issue: should conversions be 'precision preserving' (conversion towards types with the maximum number of databits) or to 'magnitude preserving' (conversion towards types with maximum number of bits in the integral part).
- make all automatic type conversion illegal: various systems will support various types of fixed point datatypes, hence automatic type conversion will behave differently on different systems. Therefore, one way of giving the programmer control over the conversions is by requiring that all conversions are indicated explicitly.

2.4.1 Mixing fixed point types with different precision

2.4.2 Mixing fixed point and integer types

2.4.3 Mixing fixed point and floating point types

2.4.4 Rounding modes, rounding control

The following rounding modes are usually supplied by the hardware when using fixed point operations:
- rounding to plus infinity
- rounding to zero
- rounding to minus infinity
- rounding to infinity
- convergent rounding
- truncation
- truncation to zero.

Question: should all these rounding modes be supported?

2.5 The semantics of shift and other operators on fixed point values

2.6 The fixed point complex type

2.7 Fixed point constants

2.8 Format conversion of fixed point types

3. Memory qualifiers

Many embedded processors have support for more than one type of (physical) memory, sometimes with overlapping address spaces, often with specific hardware instructions for access to the different types of memory. In general it is impractical, if not impossible, for a compiler to make use of, or to optimise the usage of, these various memories.

In order to give the programmer some control over the association of variables with memory, extensions (either in the form of compiler directives or as extended syntax) are necessary. It should be noted however that the semantics of such extensions is (or should be) highly dependent on the underlying hardware architecture. Hence, overspecification could easily lead to non-portable programs. It is therefore proposed to specify the absolute minimum (almost as 'stubs') with implementation defined semantics.

Possible approaches are:
- The introduction of a memory qualifier (of a class of memory qualifiers) to be used in the type specification of a variable, with implementation defined semantics.
- Binding of variable declarations to memory spaces through pragma's.

- Define a mechanism similar to the C++ *linkage specification*, using implementation defined *string literals* to indicate the memory class.

Possible discussion items:
- The relationship of the above concepts with the currently defined static, automatic and allocated storage durations (section 6.2.4): multiple stacks, parameterized malloc calls etc.
- Passing pointers to objects in various memories to 'general routines'.

4. Circular buffers

It is proposed to introduce circular arrays and circular pointers. The obvious way to do this seems to be through a new keyword **circ**: **int circ** buf[5]; and **int circ** * p = buf;. The semantics would be (roughly): an index in a circular array is always modulo the size of that array (the first element 'follows' the last element).

Some possible restrictions:
- only one dimensional arrays can be circular;
- the circular attribute cannot be inherited: only an array or pointer explicitly defined to be circular is circular;
- a variable length array cannot be circular.

5. Summary of issues, raised in this document

1. Is there a need to have keywords to distinguish between type-A and type-B fixed point datatypes?
2. Is there a need to have both **signed** and **unsigned** fixed point datatypes?
3. Is there a need to have a generalised fixed point datatype in the programming model?
4. Is there a need to have a generalised fixed point datatype to define the conversion and rounding rules?
5. If there is a need to have a generalised fixed point datatype, which one should be used?
6. What should be the spelling of the new keyword(s)? Start with an underscore?
7. Should saturation be linked to the operation or to the operands?
8. What are the usual arithmetic conversions when fixed point datatypes are involved? Are integers a subset of fixed point values which are a subset of floating point values?
9. Memory qualifiers.
10. Circular buffers.
11. Is fixed point spelled *fixed point* (as is done in this document) or as *fixed-point*?