Chapter 4

Extended and Mixed Precision BLAS

4.1 Overview

This Chapter describes *extended* and *mixed* precision implementations of the BLAS described in other chapters. Extended precision is used only internally to the BLAS; the input and output arguments remain as before. Extended precision permits us to implement some algorithms that may be simpler, more accurate, and sometimes even faster than without it. Mixed precision refers to having some input/output parameters that are both single precision and double precision, or both real and complex. Mixed precision similarly permits us to write simpler or faster algorithms. But given the complexity that could result by allowing too many combinations of types and precisions, we must choose a parsimonious subset that is both useful and reasonable to implement.

The rest of this chapter is organized as follows. Section 4.2 summarizes the designs goals and decisions that guide our design, with details left to [42]. Section 4.3 summarizes the functions supported in extended and mixed precision. This includes a discussion of the error bounds that routines must satisfy. Section 4.4 summarizes the issues in our design of language bindings for Fortran 95, Fortran 77 and C. Section 4.5 contains the detailed calling sequences for the subroutines in the three languages. A complete justification of our design appears in [42].

4.2 Design Goals and Summary

Our proposal to have extended and mixed precision in the BLAS is motivated by the following facts:

- A number of important linear algebra algorithms can become simpler, more accurate and sometimes faster if internal computations carry more precision (and sometimes more range) than is used for the input and output arguments. These include linear system solving, least squares problems, and eigenvalue problems. Often the benefits of wider arithmetic cost only a small fractional addition to the total work.
- For single precision input, the computer's native double precision is a way to achieve these benefits easily on all commercially significant computers, at least when only a few extra-precision operations are needed. (Crays and their emulators implement 64-bit single in hardware and much slower 128-bit double in software, so if a great many double precision operations are needed, these machines will slow down significantly.)
- Intel and similar processors are designed to run fastest performing arithmetic to the full 80-bit width, wider than double precision, of their internal registers. These computers confer

some benefits of wider arithmetic at little or no performance penalty. Some BLAS on these computers already perform wider arithmetic internally but, without knowing this for sure, programmers cannot exploit it.

- All computers can simulate quadruple precision or something like it in software at the cost of arithmetic slower than double precision by at worst an order of magnitude. Less slowdown is incurred for a rough double-double precision on machines (IBM RS/6000, PowerPC/Mac, SGI/MIPS R8000,HP PA RISC 2.0) with special fused multiply-accumulate instructions. Since some algorithms require very little extra precise arithmetic to get a large benefit, the slowdown is practically negligible.

Given the variety of implementation techniques hinted at above, we need to carefully examine the costs and benefits of exploiting various arithmetic features beyond the most basic ones, and choose a parsimonious subset that

Goal 1: is reasonable to implement,

Goal 2: supports some if not all important application examples,

Goal 3: is easy to use,

Goal 4: encourages the writing of portable code, and

Goal 5: accommodates growth as we learn about new algorithms exploiting our arithmetic features.

Here is an outline of our design decisions. These are discussed and justified in detail in [42].

1. We will not require that the user explicitly declare or use any new extended precision data types, i.e. beyond the standard single and double precisions, since these are not supported in a standard way by every language and compiler. Thus the only extended precision that we mandate will be hidden inside the BLAS, and so can be implemented in any convenient machine dependent way. This supports Goals 1, 3 and 4 above.

2. This internal extended precision will support most of the application examples listed in [42], supporting Goal 2.

3. Since we cannot predict all the future applications of extended or mixed precision, we will accommodate growth by making our proposal as orthogonal as possible to the rest of this proposal, showing how to take any BLAS routine, determine whether extra precision is worth using (since sometimes it is not), and define the extended precision version if it is. This supports Goal 5.

4. Since the number of possible routines with mixed precision inputs is very large, we will specify a small subset of mixed precision routines which seems to cover most foreseeable needs. This supports Goals 1 and 2.

5. In order to easily estimate error bounds in code by running with different internal precisions and then comparing the answers, (see Example 8 in [42]), we need to be able to specify the extended precision at runtime; we will do this with a variable we will call PREC. This supports Goal 2.

6. Since different machines may best support extended precision in different ways, PREC could potentially take on many machine-dependent values. Instead we have chosen a parsimonious subset that will be available on all machines, permitting the implementor to support others if desired. This supports all the Goals 1 and 4 above.

7. Since the precision specified by one value of PREC can still have different implementations and so different error bounds on different machines, we have specified environmental enquiries for the user to be able to discover the actual machine precision (or over/underflow thresholds) used at runtime. This lets the user pick appropriate stopping criteria for iterations, etc. This supports Goals 3 and 4.

4.3 Functionality

This section describes the functionality of extended and mixed precision BLAS in a language independent way. Section 4.3.1 describes how extra precision is specified via the PREC argument. Section 4.3.2 describes in general what kind of mixed precision operations will be supported. Section 4.3.3 describes the error bounds that BLAS operations must satisfy; this is where the semantics of "extra precision" are precisely specified. Finally, section 4.3.4 lists the functions that will be supported in extra and/or mixed precision.

4.3.1 Specifying Extra Precision

The internal precision to be used by an extended precision routine will be specified by an argument called PREC. It is not entirely straightforward to describe PREC because even on a single machine there may be multiple ways of implementing wider-than-double-precision arithmetic (see [42]).

To encourage portability, we specify names for precisions that may map to different formats and techniques on different machines. As discussed in section 1.6, historically the words "single" and "double" have referred to very different formats on different architectures. Nonetheless, we all agree on single precision as a word with a certain meaning, and double precision too, meaning twice or more precision than single. The definitions below add two more precisions, whose implementation details are discussed in [42].

PREC = **Single** . This means single precision, whatever single means on the particular machine, language and compiler.

PREC = **Double** . This means double precision, again whatever that means on a particular machine, language and compiler.

PREC = **Indigenous** . This means the widest hardware-supported format available. Its intention is to let the machine run close to top speed, while being as accurate as possible. On some machines this would be a 64-bit format (whether it is called single or double), but on Intel machines and ones like them it means the 80-bit IEEE format of the floating point registers.

PREC = **Extra** . This means anything at least 1.5 times as accurate than double, and in particular wider than 80-bits (see section 4.3.3 for details). An existing quadruple precision format could be used to implement this, but it can probably be implemented implemented more efficiently using native double (or indigenous) operations in a technique called "double-double", described in [42, 46, 47]. It is possible to write a portable and reasonably efficient reference implementation of all proposed routines using these techniques [42].

The actual names for PREC values are specified in section A.3. Here are the rules for using PREC:

1. The internal precision used must always be at least as high as the most precise input or output. So if the user requests less internal precision than in the most precise input or output, then the implementor must use more than requested.

2. The implementor may always use a higher precision than the one requested in the subroutine call, if this is convenient or faster.

3. The precision actually used is available to the user via the environmental enquiry in section 4.3.3.

4. PREC may take on other machine dependent values provided by the implementor, provided these are documented via the environmental enquiry routine.

Advice to implementors: While it appears that as many as seven new implementations of each routine are needed (four when the arguments are single, and three when the arguments are double), in fact fewer are needed: Two exist already as the standard BLAS (single input/output with PREC = Single, and double input/output when PREC = Double), Indigenous = Double or Indigenous = Single on many machines, and wider precision than requested may be used. Thus the only really new implementations may be single input/output with Double or Extra internal precision, and double input/output with Extra internal precision. Of these, only Extra internal precision may need arithmetic not already native to the machine. A reference implementation is described in [42].

4.3.2 Mixed Precision

Suppose a subroutine has several floating point arguments, some scalars and some arrays. Mixed precision refers to permitting these arguments to have different *mathematical types*, meaning real and complex, or different *precisions*, meaning single and double. Some BLAS in Chapter 2 are naturally defined with arguments of mixed mathematical type (e.g. HERK), but most have a single mathematical type; all are defined with the same precision for all arguments.

The permitted combinations of mathematical types and precisions are defined as follows. There are two cases:

1. The mathematical types of the input/output floating point arguments are identical to the BLAS as defined in Chapter 1. All scalar arguments and the output argument (scalar or array) are double precision. At least one array argument must be single precision.

   For example, suppose the function being implemented is matrix-matrix multiplication $C = \alpha \cdot A \cdot B + \beta \cdot C$, where α and β are scalars and A, B and C are arrays. Then the allowed types are as follows (S = Single real, D = Double real, C = Single complex, Z = Double complex).

α	A	B	β	C
D	S	S	D	D
D	S	D	D	D
D	D	S	D	D
Z	C	C	Z	Z
Z	C	Z	Z	Z
Z	Z	C	Z	Z

2. The precision of all floating point arguments must be single, or all must be double. All scalar arguments and the output argument (scalar or array) are complex (unless a scalar argument must be real for mathematical reasons, like α and β in HERK). At least one input array argument must be real.

   For example, suppose the function being implemented is matrix-matrix multiplication as before. Then the allowed types are as follows:

α	A	B	β	C
C	S	S	C	C
C	S	C	C	C
C	C	S	C	C
Z	D	D	Z	Z
Z	D	Z	Z	Z
Z	Z	D	Z	Z

Note that we specify only 16 versions of matrix-matrix multiplication (the 12 mixed ones above, and 4 unmixed), in contrast to the maximum possible $4^5 = 1024$.

4.3.3 Numerical Accuracy and Environmental Enquiries

The machine dependent interpretations of PREC require us to have a more complicated environmental enquiry routine to describe the numerical behavior of the routine in this chapter than the simpler FPINFO routine described in sections 1.6 and 2.7. While FPINFO should still be available for the user to call to get basic properties of the single and double precision floating point types, here we will specify an additional routine FPINFO_X that depends on PREC.

The calling sequence of this function is

```
result = FPINFO_X (CMACH, PREC)
```

Both arguments are input arguments, with the requested information returned as the integer value of FPINFO_X. The exact input values depend on the language, and are described in section 4.4. PREC has the same meaning as before. Input argument CMACH may take on the named constant values below, which are a subset of those permitted by function FPINFO as described in section 2.7. Only the first six values of CMACH from section 2.7 are permitted, because 1) they are sufficient to define the remaining parameters by using the formulas in section 1.6, and 2) the values returned by FPINFO_X are representable integer values, whereas the other possible return values, like the overflow and underflow thresholds, may not be representable in any user-declarable format.

Floating Point parameter	Description
BASE	base of the machine
T	number of (BASE) digits in the mantissa
RND	1 when “proper rounding” occurs in addition, 0 otherwise
IEEE	1 when rounding in addition occurs in “IEEE style”, 0 otherwise
EMIN	minimum exponent before (gradual) underflow
EMAX	largest exponent before overflow

We will use the following notation to describe machine parameters derivable from the values returned by FPINFO_X using the formulas in section 1.6:

- ϵ_{PREC} is *relative machine precision* or *machine epsilon* of the internal precision specified by PREC,
- ϵ_o is the machine epsilon for the output precision,
- $\mathrm{OV}_{\mathrm{PREC}}$ and $\mathrm{UN}_{\mathrm{PREC}}$ are the overflow and underflow thresholds for internal precision PREC, and
- OV_o and UN_o are the overflow and underflow thresholds for for the output precision.

Here are the error bounds satisfied by the extra precision routines, and how they depend on ϵ. There are two cases of interest.

1. Suppose each component of the computed result is of the form

$$r_{true} = \alpha \cdot (\sum_{i=1}^{n} a_i \cdot b_i) + \beta \cdot c \ \ ,$$

   where all quantities are scalars. This covers the dot product, scaled vector addition and scaled vector accumulation, all variants of matrix-vector and matrix-matrix products, and low-rank updates (sometimes with α and β taking on special values like zero and one). In this case the error bound, in the absence of over/underflow of any intermediate or output quantities, should satisfy

$$|r_{computed} - r_{true}| \le \gamma(n+2) \cdot \epsilon_{\mathrm{PREC}}(|\alpha| \cdot \sum_{i=1}^{n} |a_i \cdot b_i| + |\beta \cdot c|) + \epsilon_o \cdot |r_{true}| \ \ .$$

   where $\gamma = 1$ if all data is real and $\gamma = 2\sqrt{2}$ if any data is complex.

   Rationale: This accommodates all reasonable, non-Strassen based implementations, with real or complex scalars (and conventional multiplication of complex scalars), that perform all intermediate floating point operations with machine epsilon ϵ_{PREC}, with or without a guard digit, before rounding the final result to precision ϵ_o. Underflow is guaranteed to be absent if no intermediate quantity stored in precision PREC is less than $\mathrm{UN}_{\mathrm{PREC}}$ in magnitude (unless its exact value is zero) and $|r_{computed}|$ is not less than UN_o (unless its exact value is zero). Similarly, overflow is guaranteed to be absent if no intermediate quantity in precision PREC= exceeds $\mathrm{OV}_{\mathrm{PREC}}$ in magnitude, and $|r_{computed}|$ does not exceed OV_o. We avoid specifying what happens with underflow, because the implementor may reasonably choose to compute r using $\alpha \cdot (\sum a_i \cdot b_i)$, $\sum (\alpha \cdot a_i) \cdot b_i$ or $\sum a_i \cdot (\alpha \cdot b_i)$ depending on dimensions, and the error bounds in the presence of underflow can differ significantly in these three cases. See [42] for implementation recommendations and detailed error bounds in the presence of underflow.

2. Suppose the computed solution consists of one or more vectors x satisfying an n-by-n triangular system of equations

$$Tx = \alpha b$$

   where α is a scalar, b is a vector (or vectors), and T is a triangular matrix. In this case the computed solution, in the absence of over/underflow in intermediate or output quantities, satisfies

$$(T + E)(x_{computed} + e) = \alpha(b + f)$$

where $|E_{ij}| \leq \rho n \epsilon_{\mathrm{PREC}} |T_{ij}|$, $|e_i| \leq \epsilon_o |x_{computed,i}|$, $|f_i| \leq \rho n \epsilon_{\mathrm{PREC}} |b_i|$, $\rho = 1$ if all data is real, and $\rho = 6 + 4\sqrt{2}$ if any data is complex.

Rationale: This accommodates all reasonable, substitution-based methods of solution, with summations evaluated in any order, with all intermediate floating point operations done with machine epsilon ϵ_{PREC}, and with all intermediate quantities stored to the same precision. In particular, this means that the entries of $x_{computed}$ must be temporarily stored with precision ϵ_{PREC} before being rounded to the output precision at the end. Overflow and underflow are defined and treated as before. See [42] for implementation recommendations and detailed error bounds in the presence of underflow.

The values of ϵ_{PREC} must satisfy the following inequalities:

$$\begin{aligned} \epsilon_{DOUBLE} &\leq \epsilon_{SINGLE}^2 \\ \epsilon_{INDIGENOUS} &\leq \epsilon_{SINGLE} \\ \epsilon_{EXTRA} &\leq \epsilon_{DOUBLE}^{1.5} \end{aligned}$$

The first inequality says that double precision is at least twice as accurate (has twice as many significant digits) as single precision. The second inequality says that indigenous is at least as accurate as single precision. The third inequality says that extra precision is at least 1.5 times as accurate (has 1.5 times as many significant digits) as double precision.

Advice to implementors: This is only a lower bound on the number of significant digits in extra precision; most reasonable implementations can get close to twice as many digits as double precision [42]. The lower bound is intended to exclude the use of the 80-bit IEEE format as Extra precision when Double is the 64-bit IEEE format. It is important that $BASE$, T, and RND are chosen so that EPS defined by $EPS = BASE^{1-T}$ if $RND = 0$ and $EPS = .5 * BASE^{1-T}$ if $RND = 1$ can be used for error analysis. For example in the reference implementation of EXTRA precision in [42], $T = 105$ even though 106 bits are stored. Though we do not require this, the simplest way to achieve the error bounds described above is for floating operations $\odot \in \{+, -, *, /\}$ to satisfy the following bounds in the absense of over/underflow: $fl(a \odot b) = (a \odot b)(1 + \delta)$ for some $|\delta| \leq EPS$ when a and b are real, $fl(a \pm b) = (a \pm b)(1 + \delta)$ for some $|\delta| \leq \sqrt{2} \cdot EPS$ when a and b are complex, $fl(a * b) = (a * b)(1 + \delta)$ for some $|\delta| \leq 2\sqrt{2} \cdot EPS$ when a and b are complex, and $fl(a/b) = (a/b)(1 + \delta)$ for some $|\delta| \leq (6 + 4\sqrt{2}) \cdot EPS$ when a and b are complex.

The semantics of overflow and underflow are discussed more carefully in [42]; they become more complicated concepts when using implementation techniques like double-double for extra precision. The important properties they should satisfy are

1. In any precision, a quantity greater than OV generates an exception, a $\pm\infty$ symbol, or otherwise somehow indicates its complete loss of precision.

2. In any precision, the error in a floating point operation that might underflow (during some part of the calculation, if for example it is double-double) is described by $fl(a \odot b) = (a \odot b)(1 + \delta) + \eta$, for some $|\delta| \leq EPS$ and $|\eta| \leq UN$ if a and b are real, and for slighlty larger $|\delta|$ and $|\eta|$ if a and b are complex.

We choose not to specify the overflow and underflow thresholds in more detail, in order not to eliminate innovative ways of implementing extra precision.

4.3.4 Function Tables

As discussed in [42], not all BLAS routines from Chapter 2 are worth converting to extra or mixed precision, so we only include the subset that is worth converting.

Table 4.1 is a subset of Table 2.1 in Chapter 2, Reduction Operations.

Table 4.2 is a subset of Table 2.3 in Chapter 2, Vector Operations.

Table 4.3 is a subset of Table 2.5 in Chapter 2, Matrix-Vector Operations.

Table 4.4 is a subset of Table 2.7 in Chapter 2, Matrix Matrix Operations.

Dot product	$r \leftarrow \beta r + \alpha x^T y$	DOT
Sum	$r \leftarrow \sum_i x_i$	SUM

Table 4.1: Extra and Mixed Precision Reduction Operations

Scaled vector accumulation	$y \leftarrow \alpha x + \beta y,$	AXPBY
Scaled vector addition	$w \leftarrow \alpha x + \beta y$	WAXPBY

Table 4.2: Extra and Mixed Precision Vector Operations

Matrix vector product	$y \leftarrow \alpha Ax + \beta y$	GE, GB, SY, SP, SB, HE, HP, HB	MV
	$y \leftarrow \alpha A^T x + \beta y$	GE, GB	MV
	$x \leftarrow \alpha Tx,\ x \leftarrow \alpha T^T x$	TR, TB, TP	MV
Summed matrix vector multiplies	$y \leftarrow \alpha Ax + \beta Bx$	GE	SUM_MV
Triangular solve	$x \leftarrow \alpha T^{-1}x,\ x \leftarrow \alpha T^{-T}x$	TR, TB, TP	SV

Table 4.3: Extra and Mixed Precision Matrix Vector Operations

Matrix matrix product	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$	GE	MM
	$C \leftarrow \alpha AB^T + \beta C,\ C \leftarrow \alpha A^T B^T + \beta C$		
	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C$	SY, HE	MM
Triangular multiply	$B \leftarrow \alpha TB,\ B \leftarrow \alpha BT$	TR	MM
	$B \leftarrow \alpha T^T B,\ B \leftarrow \alpha BT^T$		
Triangular solve	$B \leftarrow \alpha T^{-1} B,\ B \leftarrow \alpha BT^{-1}$	TR	SM
	$B \leftarrow \alpha T^{-T} B,\ B \leftarrow \alpha BT^{-T}$		
Symmetric rank k & $2k$	$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$	SY, HE	RK
updates ($C = C^T$)	$C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$	SY, HE	R2K

Table 4.4: Extra and Mixed Precision Matrix Matrix Operations

4.4 Interface Issues

This section describes the common issues for our three language bindings: Fortran 95, Fortran 77 and C. Here is a summary of the systematic way we take a subroutine name and its argument list,

Environmental Enquiry	machine epsilon, over/underflow thresholds

Table 4.5: Environmental Enquiries for Extra and Mixed Precision Operations

and modify them to allow for extra or mixed precision:

1. **Subroutine names and mixed precision inputs.** If the language permits a subroutine argument to have more than one type, because it can dispatch the right routine based on the actual type at compile time (Fortran 95, but not Fortran 77 or C), then the subroutine name does not have to change to accommodate mixed precision. Otherwise, a new subroutine name is required, and will be created from the old one by appending characters indicating the types of the arguments.

2. **Subroutine names and extended precision.** If the language permits PREC to be an optional argument (Fortran 95, but not Fortran 77 or C), then the same subroutine name as for the non-extended precision version can be used without change. If a new name is required, it will be formed by appending _X (or _x) to the existing name. If the name has already been modified to accommodate mixed precision, _X (or _x) should be added to the end of the new name.

3. **Location of** PREC **in the calling sequence.** The new calling sequence will consist of the original calling sequence (for the BLAS routine without extra or mixed precision) with PREC appended at the end.

4. **Type of PREC.** It will be a derived type in Fortran 95, an integer in Fortran 77, and an enumerated type in C. Standard names are listed below.

5. **Environmental enquiry function.** Its output type is an integer. The input PREC is specified as above.

4.4.1 Interface Issues for Fortran 95

1. **Subroutine names and mixed precision inputs.** No new subroutine names are needed because we can exploit the optional argument interface of Fortran 95.

2. **Subroutine names and extended precision.** No new subroutine names are needed by letting PREC be an optional argument. The default in the case of no mixed precision is the standard BLAS implementation. The default in the case of mixed precision is at the discretion of the implementor, subject to the constraints of section 4.3.1.

3. **Type of** PREC. PREC is a derived type, as defined in the module `blas_operator_arguments` (see section A.4).

4. **Environmental enquiry function.** fpinfo_x(CMACH,PREC) returns an integer. PREC is as specified above. CMACH is as defined in sections 1.6, 2.7, 4.3.3, and A.4.

5. **Error Handling.** Error handling is as defined in section 2.4.6.

4.4.2 Interface Issues for Fortran 77

As described in Chapter 2, this proposal violates the letter of the ANSI Fortran 77 standard by having subroutine and variable names longer than 6 characters and with embedded underscores.

1. **Subroutine names and mixed precision inputs.** The unmodified subroutine name has a character (S, D, C or Z) that specifies the floating point argument types. This will be the type of the output argument. By applying the rules in Section 4.3.2, this also determines the types of the scalar arguments. The possible types of the remaining array arguments are listed in Section 4.3.2. The types of these arguments (written _S, _D, _C or _Z) are appended to the unmodified subroutine name, in the order in which they appear in the argument list.

   For example, consider BLAS_ZGEMM(α,A,B,β,C) (only the floating point arguments are shown). The Z in BLAS_ZGEMM means that C, α and β are all double-complex. The possible types of A and B, and the corresponding subroutine names, are:

Type of A	Type of B	Modified subroutine name
C	C	BLAS_ZGEMM_C_C
C	Z	BLAS_ZGEMM_C_Z
Z	C	BLAS_ZGEMM_Z_C
D	D	BLAS_ZGEMM_D_D
D	Z	BLAS_ZGEMM_D_Z
Z	D	BLAS_ZGEMM_Z_D

2. **Subroutine names and extended precision.** To accommodate extended precision, PREC is added as the last argument, and _X is appended to the end of subroutine name (which may already have been modified to accommodate mixed precision).

   For example, double-complex matrix-matrix multiplication implemented with extended precision is named BLAS_ZGEMM_X. Double-complex matrix-matrix multiplication where the A and B arguments are single-complex is named BLAS_ZGEMM_C_C_X.

3. **Type of** PREC. PREC is an integer (named constant), as defined in the include file `blas_namedconstants.h` (see section A.5).

4. **Environmental enquiry function.** BLAS_FPINFO_X(CMACH,PREC) returns an integer. PREC is as specified above. CMACH is as defined in sections 1.6, 2.7, 4.3.3, and A.5.

5. **Error Handling.** Error handling is as defined in section 2.5.6.

To shorten the subroutine specifications in section 4.5, we will abbreviate the list of possible subroutine names for GEMM to a single one: BLAS_xGEMM{_a_b}{_X} The prefix x may be S (single), D (double), C (complex) or Z (double complex). Also, the subroutine name may optionally be appended with _a_b, where a and b are the types of A and B respectively, and then optionally be appended with _X. At least one of _a_b or _X must appear.

4.4.3 Interface Issues for C

1. **Subroutine names and mixed precision inputs.** The same scheme is used as in Fortran 77, as described above, except that all characters in subroutine names are lower case.

2. **Subroutine names and extended precision.** The same scheme is used as in Fortran 77, as described above, except that all characters in subroutine names are lower case.

3. **Type of PREC.** PREC is an enumerated type, as defined in the include file `blas_enum.h` (see section A.6).

4. **Environmental enquiry function.** BLAS_fpinfo_x(CMACH,PREC) returns an integer. PREC is as specified above. CMACH is as defined in sections 1.6, 2.7, 4.3.3, and A.6.

5. **Error Handling.** Error handling is as defined in section 2.6.9.

4.5 Language Bindings

4.5.1 Overview

As in Chapter 2, each specification of a routine will correspond to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The specification will have the form:

NAME (*multi-word description of operation*) $< mathematical\ representation >$

- Fortran 95 binding
- Fortran 77 binding
- C binding

Section 4.4 describes abbreviations we use below. For example,

```
SUBROUTINE BLAS_xDOT{_a_b}{_X}( N, ALPHA, X, INCX, BETA,
            Y, INCY, R [, PREC])
```

means that the subroutine name may optionally be appended with _a_b, where a and b are the types of X and Y, respectively, and also optionally appended with _X, in which case the parameter PREC must also appear.

The routines specified here are

- Reduction Operations (section 4.5.2)
  - DOT (Dot product)
  - SUM (Sum)
- Vector Operations (section 4.5.3)
  - AXPBY (Scaled vector accumulation)
  - WAXPBY (Scaled vector addition)
- Matrix-Vector Operations (section 4.5.4)
  - {GE,GB}MV (Matrix vector product)
  - {SY,SB,SP}MV (Symmetric matrix vector product)

  - {HE,HB,HP}MV (Hermitian matrix vector product)
  - {TR,TB,TP}MV (Triangular matrix vector product)
  - GE_SUM_MV (Summed matrix vector multiplies)
  - {TR,TB,TP}SV (Triangular solve)

- Matrix-Matrix Operations (section 4.5.5)
  - GEMM (General Matrix Matrix product)
  - SYMM (Symmetric matrix matrix product)
  - HEMM (Hermitian matrix matrix product)
  - TRMM (Triangular matrix matrix multiply)
  - TRSM (Triangular solve)
  - SYRK (Symmetric rank-k update)
  - HERK (Hermitian rank-k update)
  - SYR2K (Symmetric rank-2k update)
  - HER2K (Hermitian rank-2k update)

4.5.2 Mixed and Extended Precision Reduction Operations

DOT (Dot Product) $x, y \in \mathbb{R}^n, r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i$

$$x, y \in \mathbb{C}^n, r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i \text{ or } r \leftarrow \beta r + \alpha x^H y = \beta r + \alpha \sum_{i=0}^{n-1} \bar{x}_i y_i$$

The routine DOT adds the scaled dot product of two vectors x and y into a scaled scalar r. The routine returns immediately if n is less than zero, or, if beta is equal to one and either alpha or n is equal to zero. If alpha is equal to zero then x and y are not read. Similarly, if beta is equal to zero, r is not read. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

When x and y are complex vectors, the vector components x_i are used unconjugated or conjugated as specified by the operator argument conj. If x and y are real vectors, the operator argument conj has no effect.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE dot( x, y, r [, conj] [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (IN) :: y(:)
  <type>(<wp>), INTENT (INOUT) :: r
  TYPE (blas_conj_type), INTENT(IN), OPTIONAL :: conj
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x and y have shape (n)
```

The types of alpha, x, y, beta and r are governed according to the rules of mixed precision arguments set down in section 4.3: the types of x and y can optionally differ from that of r, alpha and beta.

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xDOT{_a_b}{_X}( CONJ, N, ALPHA, X, INCX, BETA, Y, INCY,
$                                R, [, PREC] )
 INTEGER           CONJ, INCX, INCY, N [, PREC]
 <type>            ALPHA, BETA, R
 <type>            X( * )
 <type>            Y( * )
```

The types of ALPHA, X, Y, BETA and R are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of X and _b is the type of Y. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xdot{_a_b}{_x}( enum blas_conj_type conj, int n, SCALAR_IN alpha,
                          const ARRAY x, int incx, SCALAR_IN beta,
                          const ARRAY y, int incy, SCALAR_INOUT r,
                          [, enum blas_prec_type prec] );
```

The types of alpha, x, y, beta and r are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument x and _b is the type of argument y. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

SUM (Sum) $$r \leftarrow \sum_{i=0}^{n-1} x_i$$

The routine SUM computes the sum of the entries of a vector x. If n is less than or equal to zero, this routine returns immediately with the output scalar r set to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

Extended precision is permitted, but not mixed precision.

This routine has the same specification as in Chapter 2, except that extended precision is permitted. Mixed precision is not permitted.

- Fortran 95 binding:

```
<type>(<wp>) FUNCTION sum( x, prec )
  <type>(<wp>), INTENT (IN) :: x(:)
  TYPE (blas_prec_type), INTENT (IN) :: prec
where
  x has shape (n)
```

The types of `sum` and `x` are identical.

- Fortran 77 binding:

```
<type> FUNCTION BLAS_xSUM_X( N, X, INCX, PREC )
INTEGER            INCX, N, PREC
<type>             X( * )
```

The types of `BLAS_xSUM_X` and argument `X` are both specified by the prefix `x`.

- C binding:

```
void BLAS_xsum_x( int n, const ARRAY x, int incx, SCALAR_INOUT sum,
                  enum blas_prec_type prec );
```

The types of arguments `sum` and `x` are both specified by the prefix `x`.

4.5.3 Mixed and Extended Precision Vector Operations

AXPBY (Scaled vector accumulation) $y \leftarrow \alpha x + \beta y$

The routine AXPBY scales the vector x by α and the vector y by β, adds these two vectors to one another and stores the result in the vector y. If n is less than or equal to zero, or if α is equal to zero and β is equal to one, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

Extended and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE axpby( x, y [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (INOUT) :: y(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x and y have shape (n)
```

The default value for β is 1.0 and (1.0,0.0).

The types of `x`, `y`, `alpha`, and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `x` can optionally differ from that of `alpha`, `beta` and `y`.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAXPBY{_a}{_X}( N, ALPHA, X, INCX, BETA, Y, INCY
                                [, PREC] )
INTEGER            INCX, INCY, N [, PREC]
<type>             ALPHA, BETA
<type>             X( * )
<type>             Y( * )
```

The types of ALPHA, X, Y, and BETA are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of X. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
void BLAS_xaxpby{_a}{_x}( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                          SCALAR_IN beta, ARRAY y, int incy,
                          [, enum blas_prec_type prec] );
```

The types of alpha, x, y, and beta are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument x. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

WAXPBY (Scaled vector addition) $w \leftarrow \alpha x + \beta y$

The routine WAXPBY scales the vector x by α and the vector y by β, adds these two vectors to one another and stores the result in the vector w. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy or incw less than zero is permitted. However, if either incx or incy or incw is equal to zero, an error flag is set and passed to the error handler.

Extended and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE waxpby( x, y, w [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (IN) :: y(:)
  <type>(<wp>), INTENT (OUT) :: w(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x, y and w have shape (n)
```

The default value for β is 1.0 and (1.0,0.0).

The types of x, y, w, alpha and beta are governed according to the rules of mixed precision arguments set down in section 4.3: the types of x and y can optionally differ from that of w, alpha and beta.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xWAXPBY{_a_b}{_X}( N, ALPHA, X, INCX, BETA, Y, INCY,
     $                                  W, INCW [, PREC] )
      INTEGER            INCW, INCX, INCY, N [, PREC]
      <type>             ALPHA, BETA
      <type>             W( * )
      <type>             X( * )
      <type>             Y( * )
```

The types of X, Y, W, ALPHA and BETA are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of X and _b is the type of Y. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xwaxpby{_a_b}{_x}( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                             SCALAR_IN beta, const ARRAY y, int incy, ARRAY w,
                             int incw [, enum blas_prec_type prec] );
```

The types of x, y, w, alpha and beta are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument x and _b is the type of argument y. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

4.5.4 Mixed and Extended Precision Matrix-Vector Operations

{GE,GB}MV (Matrix vector product) $y \leftarrow \alpha A x + \beta y$, $y \leftarrow \alpha A^T x + \beta y$ or $y \leftarrow \alpha A^H x + \beta y$

The routines multiply a vector x by a general (or general band) matrix A or its transpose, or its conjugate transpose, scales the resulting vector and adds it to the scaled vector operand y. If m or n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine GEMV, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GBMV, if kl or ku is less than zero, or if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

Extended and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
    SUBROUTINE gbmv( a, m, kl, x, y [, trans] [, alpha] [, beta] [, prec] )
      <type>(<wp>), INTENT(IN) :: a(:,:), x(:)
```

```
      INTEGER, INTENT(IN) :: m, kl
      <type>(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      if trans = blas_no_trans then
         x has shape (n)
         y has shape (m)
      else if trans =/ blas_no_trans then
         x has shape (m)
         y has shape (n)
      end if
```

The functionality of gemv is covered by gemm.

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMV{_a_b}{_X}( TRANS, M, N, ALPHA, A, LDA,
     $                                 X, INCX, BETA, Y, INCY [, PREC] )
General Band:
      SUBROUTINE BLAS_xGBMV{_a_b}{_X}( TRANS, M, N, KL, KU, ALPHA, A,
     $                                 LDA, X, INCX, BETA, Y, INCY [, PREC] )
all:
      INTEGER            INCX, INCY, KL, KU, LDA, M, N, [PREC,] TRANS
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
      <type>             X( * )
      <type>             Y( * )
```

The types of ALPHA, A, X, Y, and BETA are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A and _b is the type of X. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
General:
void BLAS_xgemv{_a_b}{_x}( enum blas_order_type order,
                           enum blas_trans_type trans, int m, int n,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY x, int incx, SCALAR_IN beta, ARRAY y,
                           int incy [, enum blas_prec_type prec] );
General Band:
void BLAS_xgbmv{_a_b}{_x}( enum blas_order_type order,
                           enum blas_trans_type trans, int m, int n,
                           int kl, int ku, SCALAR_IN alpha,
                           const ARRAY a, int lda, const ARRAY x, int incx,
```

```
                              SCALAR_IN beta, ARRAY y, int incy
                              [, enum blas_prec_type prec] );
```

The types of alpha, a, x, y and beta are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a and _b is the type of argument x. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

{SY,SB,SP}MV (Symmetric matrix vector multiply) $y \leftarrow \alpha Ax + \beta y$ with $A = A^T$

The routines multiply a vector x by a real or complex symmetric matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine SYMV, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SBMV, if lda is less than k plus one, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Symmetric Band:
    SUBROUTINE sbmv( a, x, y [, uplo] [, alpha] [, beta] [, prec] )
Symmetric Packed:
    SUBROUTINE spmv( ap, x, y [, uplo] [, alpha] [, beta] [, prec] )
      <type>(<wp>), INTENT(IN) :: <aa>
      <type>(<wp>), INTENT(IN) :: x(:)
      <type>(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <aa> ::= a(:,:) or ap(:)
    and
      SB  a has shape (k+1,n)
      SP  ap has shape (n*(n+1)/2)
      x and y have shape (n)
```

The types of alpha, a or ap, x, beta, and y are governed by the rules of mixed precision arguments set down in section 4.3: the types of a or ap and x can optionally differ from that of y, alpha and beta.

The functionality of symv is covered by symm.

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYMV{_a_b}{_X}( UPLO, N, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Symmetric Band:
      SUBROUTINE BLAS_xSBMV{_a_b}{_X}( UPLO, N, K, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Symmetric Packed:
      SUBROUTINE BLAS_xSPMV{_a_b}{_X}( UPLO, N, ALPHA, AP, X, INCX, BETA,
     $                                 Y, INCY [, PREC] )
all:
      INTEGER            INCX, INCY, K, LDA, N, UPLO [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * )
      <type>             X( * )
      <type>             Y( * )
```

The types of `ALPHA`, `A` or `AP`, `X`, `Y` and `BETA` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` or `AP`, and `_b` is the type of `X`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
Symmetric:
void BLAS_xsymv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY x, int incx, SCALAR_IN beta, ARRAY y,
                           int incy [, enum blas_prec_type prec] );
Symmetric Band:
void BLAS_xsbmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, int k, SCALAR_IN alpha, const ARRAY a,
                           int lda, const ARRAY x, int incx, SCALAR_IN beta,
                           ARRAY y, int incy [, enum blas_prec_type prec] );
Symmetric Packed:
void BLAS_xspmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, SCALAR_IN alpha, const ARRAY ap,
                           const ARRAY x, int incx, SCALAR_IN beta, ARRAY y,
                           int incy [, enum blas_prec_type prec] );
```

The types of `alpha`, `a` or `ap`, `x`, `y`, and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a` or `ap` and `_b` is the type of argument `x`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

{HE,HB,HP}MV (Hermitian matrix vector product) $y \leftarrow \alpha A x + \beta y$ with $A = A^H$

The routines multiply a vector x by a Hermitian matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. The imaginary part of the diagonal entries of the matrix operand are supposed to be zero and should not be referenced. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine HEMV, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine HBMV, if lda is less than k plus one, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Hermitian Band:
    SUBROUTINE hbmv{_a}{_x}( a, x, y  [, uplo] [, alpha] [, beta] [, prec] )
Hermitian Packed:
    SUBROUTINE hpmv{_a}{_x}( ap, x, y [, uplo] [, alpha] [, beta] [, prec] )
      COMPLEX(<wp>), INTENT(IN) :: <aa>
      COMPLEX(<wp>), INTENT(IN) :: x(:)
      COMPLEX(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <aa>  ::= a(:,:) or ap(:)
    and
      HB  a has shape (k+1,n)
      HP  ap has shape (n*(n+1)/2)
      x and y have shape (n)
```

The types of alpha, a or ap, x, beta, and y are governed by the rules of mixed precision arguments set down in section 4.3: the types of a or ap and x can optionally differ from that of y, alpha and beta.

The functionality of hemv is covered by hemm.

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHEMV{_a_b}{_X}( UPLO, N, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Hermitian Band:
      SUBROUTINE BLAS_xHBMV{_a_b}{_X}( UPLO, N, K, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Hermitian Packed:
      SUBROUTINE BLAS_xHPMV{_a_b}{_X}( UPLO, N, ALPHA, AP, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
all:
```

```
INTEGER            INCX, INCY, K, LDA, N, UPLO [, PREC]
<ctype>            ALPHA, BETA
<ctype>            A( LDA, * ) or AP( * )
<ctype>            X( * )
<ctype>            Y( * )
```

The types of ALPHA, A or AP, X, Y, and BETA are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A or AP and _b is the type of X. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
Hermitian:
void BLAS_xhemv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, CSCALAR_IN alpha, const CARRAY a, int lda,
                           const CARRAY x, int incx, CSCALAR_IN beta, CARRAY y,
                           int incy [, enum blas_prec_type prec] );
Hermitian Band:
void BLAS_xhbmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, int k, CSCALAR_IN alpha, const CARRAY a,
                           int lda, const CARRAY x, int incx, CSCALAR_IN beta,
                           CARRAY y, int incy [, enum blas_prec_type prec] );
Hermitian Packed:
void BLAS_xhpmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, CSCALAR_IN alpha, const CARRAY ap,
                           const CARRAY x, int incx, CSCALAR_IN beta, CARRAY y,
                           int incy [, enum blas_prec_type prec] );
```

The types of alpha, a or ap, x, y, and beta are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a or ap and _b is the type of argument x. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

{TR,TB,TP}MV (Triangular matrix vector product) $x \leftarrow \alpha T x$, $x \leftarrow \alpha T^T x$ or $x \leftarrow \alpha T^H x$

The routines multiply a vector x by a general triangular matrix T or its transpose, or its conjugate transpose, and copies the resulting vector in the vector operand x. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. For the routine TRMV, if ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler. For the routine TBMV, if ldt is less than k plus one, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Triangular Band:
    SUBROUTINE tbmv( t, x  [, uplo] [, transt] [, diag] [, alpha] [, prec] )
Triangular Packed:
    SUBROUTINE tpmv( tp, x [, uplo] [, transt] [, diag] [, alpha] [, prec] )
      <type>(<wp>), INTENT(IN) :: <tt>
      <type>(<wp>), INTENT(INOUT) :: x(:)
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <tt>  ::= t(:,:) or tp(:)
    and
      x has shape (n)
      TB  t has shape (k+1,n)
      TP  tp has shape (n*(n+1)/2)
    (k=band width)
```

The types of `alpha`, `t` or `tp`, and `x` are governed by the rules of mixed precision arguments set down in section 4.3: the type of `t` or `tp` can optionally differ from that of `x` and `alpha`.

The functionality of trmv is covered by trmm.

- Fortran 77 binding:

```
Triangular:
       SUBROUTINE BLAS_xTRMV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
      $                               INCX [, PREC] )
Triangular Band:
       SUBROUTINE BLAS_xTBMV{_a}{_X}( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT,
      $                               X, INCX [, PREC] )
Triangular Packed:
       SUBROUTINE BLAS_xTPMV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX
      $                               [, PREC] )
all:
      INTEGER            DIAG, INCX, K, LDT, N, TRANS, UPLO [, PREC]
      <type>             ALPHA
      <type>             T( LDT, * ) or TP( * )
      <type>             X( * )
```

The types of `ALPHA`, `T` or `TP`, and `X` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `T` or `TP`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a` and `_X` must be present.

- C binding:

```
Triangular:
void BLAS_xtrmv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Band:
void BLAS_xtbmv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, int k, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Packed:
void BLAS_xtpmv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY tp,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
```

The types of `alpha`, `t` or `tp`, and `x` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of argument `t` or `tp`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a` and `_x` must be present.

GE_SUM_MV (Summed matrix vector multiplies) $y \leftarrow \alpha Ax + \beta Bx$

This routine adds the product of two scaled matrix vector products. It can be used to compute the residual of an approximate eigenvector and eigenvalue of the generalized eigenvalue problem $A * x = \lambda * B * x$. If m or n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, or ldb is less than one or ldb is less than m, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE ge_sum_mv( a, x, b, y [, alpha] [, beta] [, prec])
  <type>(<wp>), INTENT (IN) :: a(:,:), b(:,:)
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (OUT) :: y(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  <type>(blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x has shape (n);
  y has shape (m);
  a and b have shape (m,n) for general matrices
```

The types of `alpha`, `a`, `x`, `beta`, `b`, and `y` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from that of `x`, `y`, `alpha` and `beta`. Arguments `a` and `b` must have the same type.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGE_SUM_MV{_a_b}{_X}( M, N, ALPHA, A, LDA, X, INCX,
     $                                     BETA, B, LDB, Y, INCY
     $                                     [, PREC] )
      INTEGER            INCX, INCY, LDA, LDB, M, N [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), B( LDB, * )
      <type>             X( * )
      <type>             Y( * )
```

The types of `ALPHA`, `A`, `X`, `BETA`, `B`, and `Y` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `A` and `B`, and `_b` is the type of `x`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xge_sum_mv{_a_b}{_x}( enum blas_order_type order, int m, int n,
                                SCALAR_IN alpha, const ARRAY a, int lda,
                                const ARRAY x, int incx, SCALAR_IN beta,
                                const ARRAY B, int ldb, ARRAY y, int incy
                                [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `x`, `beta`, `b`, and `y` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `a` and `b`, and `_b` is the type of `x`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

{TR,TB,TP}SV (Triangular solve) $x \leftarrow \alpha T^{-1} x$, $x \leftarrow \alpha T^{-T} x$

These functions solve one of the systems of equations $x \leftarrow \alpha T^{-1} x$ or $y \leftarrow \alpha T^{-1} x$, where x and y are vectors and the matrix T is a unit, non-unit, upper or lower triangular (or triangular banded or triangular packed) matrix. If n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

Advice to implementors. Note that no check for singularity, or near singularity is specified for these triangular equation-solving functions. The requirements for such a test depend on the application, and so we felt that this should not be included, but should instead be performed before calling the triangular solver.

To implement this function when the internal precision requested is higher than the precision of x, temporary workspace is needed to compute and store x internally to higher precision. (*End of advice to implementors.*)

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Triangular Band:
    SUBROUTINE tbsv( t, x [, uplo] [, transt] [, diag] [, alpha] [, prec] )
Triangular Packed:
    SUBROUTINE tpsv( tp, x [, uplo] [, trans] [, diag] [, alpha] [, prec] )
      <type>(<wp>), INTENT(IN) :: <tt>
      <type>(<wp>), INTENT(INOUT) :: x(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <tt>  ::= t(:,:) or tp(:)
    and
      x has shape (n)
      TB  t has shape (k+1,n)
      TP  tp has shape (n*(n+1)/2)
    (k=band width)
```

The types of alpha, t or tp, and x are governed by the rules of mixed precision arguments set down in section 4.3: the type of t or tp can optionally differ from that of x and alpha.

The functionality of trsv is covered by trsm.

- Fortran 77 binding:

```
Triangular:
       SUBROUTINE BLAS_xTRSV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, T, LDT,
      $                              X, INCX [, PREC] )
Triangular Band:
       SUBROUTINE BLAS_xTBSV{_a}{_X}( UPLO, TRANS, DIAG, N, K, ALPHA, T,
      $                              LDT, X, INCX [, PREC] )
Triangular Packed:
       SUBROUTINE BLAS_xTPSV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, TP, X,
      $                              INCX [, PREC] )
all:
      INTEGER           DIAG, INCX, K, LDT, N, TRANS, UPLO [, PREC]
      <type>            ALPHA
      <type>            T( LDT, * ) or TP( * )
      <type>            X( * )
```

The types of ALPHA, T or TP, and X are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of T or TP. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
Triangular:
void BLAS_xtrsv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Band:
void BLAS_xtbsv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, int k, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Packed:
void BLAS_xtpsv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY tp, ARRAY x,
                         int incx [, enum blas_prec_type prec] );
```

The types of `alpha`, `t` or `tp`, and `x` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of argument `t` or `tp`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a` and `_x` must be present.

4.5.5 Mixed and Extended Precision Matrix-Matrix Operations

In the following section, $op(X)$ denotes X, or X^T or X^H where X is a matrix.

GEMM (General Matrix Matrix Product) $C \leftarrow \alpha op(A)op(B) + \beta C$

The routine performs a general matrix matrix multiply $C \leftarrow \alpha op(A)op(B) + \beta C$ where α and β are scalars, and A, B, and C are general matrices. This routine returns immediately if m or n or k is less than or equal to zero. If lda is less than one or less than m, or if ldb is less than one or less than k, or if ldc is less than one or less than m, an error flag is set and passed to the error handler.

This interface encompasses the Legacy BLAS routine xGEMM.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE gemm( a, b, c [, transa] [, transb] [, alpha] [, beta] &
                 [, prec] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(IN) :: <bb>
  <type>(<wp>), INTENT(INOUT) :: <cc>
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa, transb
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
```

```
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
  <cc>  ::= c(:,:) or c(:)
and
  c, rank 2, has shape (m,n)
       a has shape (m,k) if transa = blas_no_trans (the default)
                   (k,m) if transa /= blas_no_trans
                   (m) if rank 1
       b has shape (k,n) if transb = blas_no_trans (the default)
                   (n,k) if transb /= blas_no_trans
                   (n) if rank 1
  c, rank 1, has shape (m)
       a has shape (m,n) if transa = blas_no_trans (the default)
                   (n,m) if transa /= blas_no_trans
       b has shape (n)
```

Rank a	Rank b	Rank c	transa	transb	Operation	Arguments
2	2	2	N	N	$C \leftarrow \alpha AB + \beta C$	real or complex
2	2	2	N	T	$C \leftarrow \alpha AB^T + \beta C$	real or complex
2	2	2	N	H	$C \leftarrow \alpha AB^H + \beta C$	complex
2	2	2	T	N	$C \leftarrow \alpha A^T B + \beta C$	real or complex
2	2	2	T	T	$C \leftarrow \alpha A^T B + \beta C$	real or complex
2	2	2	H	N	$C \leftarrow \alpha A^H B + \beta C$	complex
2	2	2	H	H	$C \leftarrow \alpha A^H B^H + \beta C$	complex
2	1	1	N	-	$c \leftarrow \alpha Ab + \beta c$	real or complex
2	1	1	T	-	$c \leftarrow \alpha A^T b + \beta c$	real or complex
2	1	1	H	-	$c \leftarrow \alpha A^H b + \beta c$	complex
1	1	2	-	-	$C \leftarrow \alpha ab^T + \beta C$	real or complex
1	1	2	-	H	$C \leftarrow \alpha ab^H + \beta C$	complex

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xGEMV is also covered by this generic procedure.

The types of `a`, `b`, `c`, `alpha` and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from that of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMM{_a_b}{_X}( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
     $                                 B, LDB, BETA, C, LDC [, PREC] )
      INTEGER             K, LDA, LDB, LDC, M, N, TRANSA, TRANSB [, PREC]
      <type>              ALPHA, BETA
      <type>              A( LDA, * )
```

```
      <type>           B( LDB, * )
      <type>           C( LDC, * )
```

The types of `ALPHA`, `A`, `B`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` and `_b` is the type of `B`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xgemm{_a_b}{_x}( enum blas_order_type order,
                           enum blas_trans_type transa,
                           enum blas_trans_type transb, int m, int n, int k,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY b, int ldb,
                           SCALAR_IN beta, ARRAY c, int ldc
                           [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a` and `_b` is the type of argument `b`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

SYMM (Symmetric Matrix Matrix Product) $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

This routine performs one of the symmetric matrix matrix operations $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where α and β are scalars, A is a symmetric matrix, and B and C are general matrices. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

The interfaces encompass the Legacy BLAS routine xSYMM with added functionality for complex symmetric matrices.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
    SUBROUTINE symm( a, b, c [, side] [, uplo] [, alpha] [, beta] [, prec] )
      <type>(<wp>), INTENT(IN) :: a(:,:)
      <type>(<wp>), INTENT(IN) :: <bb>
      <type>(<wp>), INTENT(INOUT) :: <cc>
      TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
```

```
   TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
 where
   <bb>  ::= b(:,:) or b(:)
   <cc>  ::= c(:,:) or c(:)
 and
  c, rank 2, has shape (m,n), b same shape as c
    SY  a has shape (m,m) if side = blas_left_side (the default)
        a has shape (n,n) if side /= blas_left_side
  c, rank 1, has shape (m), b same shape as c
    SY  a has shape (m,m)
```

Rank b	Rank c	side	Operation
2	2	L	$C \leftarrow \alpha AB + \beta C$
2	2	R	$C \leftarrow \alpha BA + \beta C$
1	1	-	$c \leftarrow \alpha Ab + \beta c$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xSYMV is covered by symm.

The types of a, b, c, alpha and beta are governed according to the rules of mixed precision arguments set down in section 4.3: the types of a and b can optionally differ from that of c, alpha and beta.

- Fortran 77 binding:

```
   SUBROUTINE BLAS_xSYMM{_a_b}{_X}( SIDE, UPLO, M, N, ALPHA, A, LDA,
  $                                B, LDB, BETA, C, LDC [, PREC] )
   INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO [, PREC]
   <type>             ALPHA, BETA
   <type>             A( LDA, * )
   <type>             B( LDB, * )
   <type>             C( LDC, * )
```

The types of ALPHA, A, B, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A and _b is the type of B. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xsymm{_a_b}{_x}( enum blas_order_type order,
                           enum blas_side_type side,
                           enum blas_uplo_type uplo, int m, int n,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY b, int ldb, SCALAR_IN beta, ARRAY c,
                           int ldc [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a`, and `_b` is the type of argument `b`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

HEMM (Hermitian Matrix Matrix Product) $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

This routine performs one of the Hermitian matrix matrix operations $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where α and β are scalars, A is a Hermitian matrix, and B and C are general matrices. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

The interfaces encompass the Legacy BLAS routine xHEMM.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Hermitian:
    SUBROUTINE hemm( a, b, c [, side] [, uplo] [, alpha] [, beta] [, prec] )
      COMPLEX(<wp>), INTENT(IN) :: a(:,:)
      COMPLEX(<wp>), INTENT(IN) :: <bb>
      COMPLEX(<wp>), INTENT(INOUT) :: <cc>
      TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
    where
      <bb>  ::= b(:,:) or b(:)
      <cc>  ::= c(:,:) or c(:)
    and
     c, rank 2, has shape (m,n), b same shape as c
       HE  a has shape (m,m) if "side" = blas_left_side (the default)
           a has shape (n,n) if "side" /= blas_left_side
     c, rank 1, has shape (m), b same shape as c
       HE  a has shape (m,m)
```

Rank `b`	Rank `c`	`side`	Operation
2	2	L	$C \leftarrow \alpha AB + \beta C$
2	2	R	$C \leftarrow \alpha BA + \beta C$
1	1	-	$c \leftarrow \alpha Ab + \beta c$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xHEMV is covered by hemm.

The types of a, b, c, alpha and beta are governed according to the rules of mixed precision arguments set down in section 4.3: the types of a and b can optionally differ from that of c, alpha and beta.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHEMM{_a_b}{_X}( SIDE, UPLO, M, N, ALPHA, A, LDA,
     $                                 B, LDB, BETA, C, LDC [, PREC] )
      INTEGER             LDA, LDB, LDC, M, N, SIDE, UPLO [, PREC]
      <ctype>             ALPHA, BETA
      <ctype>             A( LDA, * )
      <ctype>             B( LDB, * )
      <ctype>             C( LDC, * )
```

The types of ALPHA, A, B, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A and _b is the type of B. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xhemm{_a_b}{_x}( enum blas_order_type order,
                           enum blas_side_type side,
                           enum blas_uplo_type uplo, int m, int n,
                           CSCALAR_IN alpha, const CARRAY a, int lda,
                           const CARRAY b, int ldb, CSCALAR_IN beta, CARRAY c,
                           int ldc [, enum blas_prec_type prec] );
```

The types of alpha, a, b, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a, and _b is the type of argument b. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

TRMM (Triangular Matrix Matrix Multiply) $B \leftarrow \alpha op(T)B$ or $B \leftarrow \alpha Bop(T)$

These routines perform one of the matrix-matrix operations $B \leftarrow \alpha op(T)B$ or $B \leftarrow \alpha Bop(T)$ where α is a scalar, B is a general matrix, and T is a unit, or non-unit, upper or lower triangular matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xTRMM.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE trmm( t, b [, side] [, uplo] [, transt] [, diag] &
                 [, alpha] [, prec] )
  <type>(<wp>), INTENT(IN) :: t(:,:)
  <type>(<wp>), INTENT(INOUT) :: <bb>
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
  TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <bb>  ::= b(:,:) or b(:)
and
 b, rank 2, has shape (m,n)
   TR  t has shape (m,m) if side = blas_left_side (the default)
       t has shape (n,n) if side /= blas_left_side
 b, rank 1, has shape (m)
   TR  t has shape (m,m)
```

Rank b	transa	side	Operation
2	N	L	$B \leftarrow \alpha TB$
2	T	L	$B \leftarrow \alpha T^T B$
2	H	L	$B \leftarrow \alpha T^H B$
2	N	R	$B \leftarrow \alpha BT$
2	T	R	$B \leftarrow \alpha BT^T$
2	H	R	$B \leftarrow \alpha BT^H$
1	N	-	$b \leftarrow \alpha Tb$
1	T	-	$b \leftarrow \alpha T^T b$
1	H	-	$b \leftarrow \alpha T^H b$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xTRMV is covered by `trmm`.

The types of `alpha`, `t`, and `b` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `t` can optionally differ from that of `b` and `alpha`.

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xTRMM{_a}{_X}( SIDE, UPLO, TRANST, DIAG, M, N,
 $                              ALPHA, T, LDT, B, LDB [, PREC] )
  INTEGER            DIAG, LDT, LDB, M, N, SIDE, TRANST, UPLO
 $                   [, PREC]
  <type>             ALPHA
  <type>             T( LDT, * )
  <type>             B( LDB, * )
```

The types of ALPHA, T, and B are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of T. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
void BLAS_xtrmm{_a}{_x}(enum blas_order_type order, enum blas_side_type side,
                        enum blas_uplo_type uplo, enum blas_trans_type transa,
                        enum blas_diag_type diag, int m, int n,
                        SCALAR_IN alpha, const ARRAY t, int ldt, ARRAY b,
                        int ldb [, enum blas_prec_type prec] );
```

The types of alpha, t, and b are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument t. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

TRSM (Triangular Solve) $B \leftarrow \alpha op(T^{-1})B$ or $B \leftarrow \alpha Bop(T^{-1})$

This routine solves one of the matrix equations $B \leftarrow \alpha op(T^{-1})B$ or $B \leftarrow \alpha Bop(T^{-1})$ where α is a scalar, B is a general matrix, and T is a unit, or non-unit, upper or lower triangular matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xTRSM.

Extended precision and mixed precision are permitted.

Advice to implementors. Note that no check for singularity, or near singularity is specified for these triangular equation-solving functions. The requirements for such a test depend on the application, and so we felt that this should not be included, but should instead be performed before calling the triangular solver.

To implement this function when the internal precision requested is higher than the precision of B, temporary workspace is needed to compute and store B internally to higher precision. (*End of advice to implementors.*)

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE trsm( t, b [, side] [, uplo] [, transt] [, diag] &
                 [, alpha] [, prec] )
  <type>(<wp>), INTENT(IN) :: t(:,:)
  <type>(<wp>), INTENT(INOUT) :: <bb>
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
```

```
        TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
        TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
        TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
        <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
        TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
      where
        <bb>  ::= b(:,:) or b(:)
      and
       b, rank 2, has shape (m,n)
         TR  t has shape (m,m) if side = blas_left_side (the default)
             t has shape (n,n) if side /= blas_left_side
       b, rank 1, has shape (m)
         TR  t has shape (m,m)
```

Rank `b`	`transa`	`side`	Operation
2	N	L	$B \leftarrow \alpha T^{-1} B$
2	T	L	$B \leftarrow \alpha T^{-T} B$
2	H	L	$B \leftarrow \alpha T^{-H} B$
2	N	R	$B \leftarrow \alpha B T^{-1}$
2	T	R	$B \leftarrow \alpha B T^{-T}$
2	H	R	$B \leftarrow \alpha B T^{-H}$
1	N	-	$b \leftarrow \alpha T^{-1} b$
1	T	-	$b \leftarrow \alpha T^{-T} b$
1	H	-	$b \leftarrow \alpha T^{-H} b$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xTRSV is covered by `trsm`.

The types of `t`, `x` and `alpha` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `t` can optionally differ from that of `x` and `alpha`.

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xTRSM{_a}{_X}( SIDE, UPLO, TRANST, DIAG, M, N,
      $                                ALPHA, T, LDT, B, LDB [, PREC] )
        INTEGER            DIAG, LDT, LDB, M, N, SIDE, TRANST, UPLO
      $                    [, PREC]
        <type>             ALPHA
        <type>             T( LDT, * )
        <type>             B( LDB, * )
```

The types of `ALPHA`, `T`, and `B` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `T`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a` and `_X` must be present.

- C binding:

```
void BLAS_xtrsm{_a}{_x}( enum blas_order_type order, enum blas_side_type side,
                         enum blas_uplo_type uplo, enum blas_trans_type transt,
                         enum blas_diag_type diag, int m, int n,
                         SCALAR_IN alpha, const ARRAY t, int ldt, ARRAY b,
                         int ldb [, enum blas_prec_type prec] );
```

The types of alpha, t, and b are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument t. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

SYRK (Symmetric Rank K update) $C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^T A + \beta C$

This routine performs one of the symmetric rank k operations $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$ where α and β are scalars, C is a symmetric matrix, and A is a general matrix. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xSYRK with added functionality for complex symmetric matrices.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE syrk( a, c [, uplo] [, trans] [, alpha] [, beta] &
                 [, prec] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
and
 c has shape (n,n)
 a has shape (n,k) if trans = blas_no_trans (the default)
             (k,n) if trans /= blas_no_trans
             (n) if rank 1
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AA^T + \beta C$
2	T	$C \leftarrow \alpha A^T A + \beta C$
1	-	$C \leftarrow \alpha aa^T + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3: the type of a can optionally differ from those of c, alpha and beta.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYRK{_a}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                               C, LDC [, PREC] )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
      <type>             C( LDC, * )
```

The types of ALPHA, A, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of A. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
void BLAS_xsyrk{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, int n, int k,
                         SCALAR_IN alpha, const ARRAY a, int lda,
                         SCALAR_IN beta, ARRAY c, int ldc
                         [, enum blas_prec_type prec] );
```

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument a. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

HERK (Hermitian Rank K update) $C \leftarrow \alpha AA^H + \beta C$, $C \leftarrow \alpha A^H A + \beta C$

This routine performs one of the Hermitian rank k operations $C \leftarrow \alpha AA^H + \beta C$ or $C \leftarrow \alpha A^H A + \beta C$ where α and β are scalars, C is a Hermitian matrix, and A is a general matrix. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xHERK.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE herk( a, c [, uplo] [, trans] [, alpha] [, beta] &
                 [, prec] )
  COMPLEX(<wp>), INTENT(IN) :: <aa>
  COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  REAL(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
and
 c has shape (n,n)
 a has shape (n,k) if trans = blas_no_trans (the default)
             (k,n) if trans /= blas_no_trans
             (n) if rank 1
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AA^H + \beta C$
2	T	$C \leftarrow \alpha A^H A + \beta C$
1	-	$C \leftarrow \alpha aa^H + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of `alpha`, `a`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `a` can optionally differ from those of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHERK{_a}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                               C, LDC [, PREC] )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO [, PREC]
      <rtype>            ALPHA, BETA
      <ctype>            A( LDA, * )
      <ctype>            C( LDC, * )
```

The types of `ALPHA`, `A`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `A`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a` and `_X` must be present.

- C binding:

```
void BLAS_xherk{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, int n, int k,
                         RSCALAR_IN alpha, const CARRAY a, int lda,
                         RSCALAR_IN beta, CARRAY c, int ldc
                         [, enum blas_prec_type prec] );
```

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument a. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

SYR2K (Symmetric rank 2k update) $C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$

$C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$

These routines perform the symmetric rank 2k operation $C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$ or $C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$ where α and β are scalars, C is a symmetric matrix, and A and B are general matrices. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xSYR2K with added functionality for complex symmetric matrices.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE syr2k( a, b, c [, uplo] [, trans] [, alpha] [, beta]
                  [, prec] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(IN) :: <bb>
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
and
 c has shape (n,n)
 if trans = blas_no_trans (the default)
    a has shape (n,k)
    b has shape (n,k)
 if trans /= blas_no_trans
    a has shape (k,n)
    b has shape (k,n)
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
2	2	T	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
1	1	-	$C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from those of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYR2K{_a_b}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA,
     $                                  B, LDB, BETA, C, LDC [, PREC] )
      INTEGER           K, LDA, LDB, LDC, N, TRANS, UPLO [, PREC]
      <type>            ALPHA, BETA
      <type>            A( LDA, * )
      <type>            B( LDB, * )
      <type>            C( LDC, * )
```

The types of `ALPHA`, `A`, `B`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if _a_b is present then _a is the type of `A` and _b is the type of `B`. The suffix _X is present if and only if `PREC` is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xsyr2k{_a_b}{_x}( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans, int n, int k,
                            SCALAR_IN alpha, const ARRAY a, int lda,
                            const ARRAY b, int ldb,
                            SCALAR_IN beta, ARRAY c, int ldc
                            [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if _a_b is present then _a is the type of argument `a` and _b is the type of argument `b`. The suffix _x is present if and only if `prec` is present. One or both of the suffixes _a_b and _x must be present.

HER2K (Hermitian rank 2k update) $C \leftarrow (\alpha A)B^H + B(\alpha A)^H + \beta C$

$$C \leftarrow (\alpha A)^H B + B^H(\alpha A) + \beta C$$

These routines perform the Hermitian rank 2k operation $C \leftarrow (\alpha A)B^H + B(\alpha A)^H + \beta C$ or $C \leftarrow (\alpha A)^H B + B^H(\alpha A) + \beta C$ where α and β are scalars, C is a Hermitian matrix, and A and B are general matrices. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xHER2K.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE her2k( a, b, c [, uplo] [, trans] [, alpha] [, beta]
                  [, prec] )
  COMPLEX(<wp>), INTENT(IN) :: <aa>
  COMPLEX(<wp>), INTENT(IN) :: <bb>
  COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha
  REAL(<wp>), INTENT(IN), OPTIONAL :: beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
and
 c has shape (n,n)
 a and b have shape (n,k) if trans = blas_no_trans (the default)
                    (k,n) if trans /= blas_no_trans
                    (n) if rank 1
```

Rank `a`	Rank `b`	`trans`	Operation
2	2	N	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C$
2	2	T	$C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C$
1	1	-	$C \leftarrow \alpha ab^H + \bar{\alpha} ba^H + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from those of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xHER2K{_a_b}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA,
 $                                  B, LDB, BETA, C, LDC [, PREC] )
  INTEGER            K, LDA, LDB, LDC, N, TRANS, UPLO [, PREC]
  <ctype>            ALPHA
  <rtype>            BETA
  <ctype>            A( LDA, * )
  <ctype>            B( LDB, * )
  <ctype>            C( LDC, * )
```

The types of `ALPHA`, `A`, `B`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` and `_b` is the type of `B`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xher2k{_a_b}{_x}( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans, int n, int k,
                            CSCALAR_IN alpha, const CARRAY A, int lda,
                            const CARRAY b, int ldb,
                            RSCALAR_IN beta, CARRAY c, int ldc
                            [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a` and `_b` is the type of argument `b`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

4.5.6 Environmental Enquiry

FPINFO_X (Environmental enquiry)

This routine queries for machine-specific floating point characteristics.

- Fortran 95 binding:

```
INTEGER FUNCTION fpinfo_x( cmach, prec )
  TYPE (blas_cmach_type), INTENT (IN) :: cmach
  TYPE (blas_prec_type), INTENT (IN) :: prec
```

- Fortran 77 binding:

```
INTEGER FUNCTION BLAS_FPINFO_X( cmach, prec )
INTEGER          cmach, prec
```

- C binding:

```
int BLAS_fpinfo_x( enum blas_cmach_type cmach,
                   enum blas_prec_type prec );
```
