# A Fortran-90 Based Multiprecision System

David H. Bailey

RNR Technical Report RNR-94-013
January 6, 1995

**Abstract**

The author has developed a new version of his Fortran multiprecision computation system that is based on the Fortran-90 language. With this new approach, a translator program is not required — translation of Fortran code for multiprecision is accomplished by merely utilizing advanced features of Fortran-90, such as derived data types and operator extensions. This approach results in more reliable translation and also permits programmers of multiprecision applications to utilize the full power of the Fortran-90 language.

Three multiprecision datatypes are supported in this system: multiprecision integer, real and complex. All the usual Fortran conventions for mixed mode operations are supported, and many of the Fortran intrinsics, such as `SIN`, `EXP` and `MOD`, are supported with multiprecision arguments.

This paper also briefly describes an interesting application of this software, wherein new number-theoretic identities have been discovered by means of multiprecision computations.

Author's address: NAS Scientific Computation Branch, NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000; `dbailey@nas.nasa.gov`.

## 1. Introduction

Readers may be familiar with the author's previous multiprecision system [3], which consists of the TRANSMP translator program and the MPFUN package of multiprecision (MP) computation routines. Together they permit one to write straightforward Fortran-77 code that can be executed using an arbitrarily high level of numeric precision.

From its inception, the TRANSMP program was intended only as an interim tool until Fortran-90 was available. This is because advanced Fortran-90 features such as derived data types and operator extensions permit one to implement multiprecision translation in a much more natural way. Now that day has arrived — Fortran-90 is currently available on several computer systems, and it soon will be available from all major vendors of scientific computers. Accordingly, the author has written a set of Fortran-90 modules that permit the user to handle MP data like any other Fortran data type.

With the new Fortran-90 based system, one declares variables to be of type MP integer, MP real or MP complex using Fortran-90 type statements. With a few exceptions, one can then write ordinary Fortran-90 code involving these variables. In particular, arithmetic operations involving these variables are performed with a numeric precision level that can be set to an arbitrarily high level. Also, most of the Fortran intrinsic functions, such as `SIN, EXP` and `MOD`, are defined with MP arguments.

In comparison to the TRANSMP approach, there are a few disappointments. To begin with, one has to give up the ability to run MP source code, without change, as a standard single precision or double precision program. Also, features such as read/write statements are not as elegant in the new system — subroutines must now be called for formatted MP read and write.

On the other hand, features such as generic functions work much better in the Fortran-90 version. Also, the coverage of Fortran features is more complete with the Fortran-90 version than with TRANSMP — programmers can now utilize the full power of the Fortran-90 language in a MP application. Another important advantage of the Fortran-90 approach is that a very reliable translation is produced, since the process of translation is performed by the Fortran-90 compiler itself, rather than by the TRANSMP program.

This article gives an overview of this new software, including a brief summary of the instructions for usage. It also describes an interesting application of this software to mathematical number theory, showing how MP calculations can be used to discover new mathematical identities.

This software is available by sending electronic mail to `mp-request@nas.nasa.gov`. Include `send index` as either the subject line or the text of the first message to this address. It is also available by using Mosaic software at the address
`http://www.nas.nasa.gov/RNR/software.html`.

## 2. The Fortran-90 MP Translation Modules

The new MP translator is a set of Fortran-90 modules. These translation modules serve as a link between the user's program and MPFUN, the library of MP computation routines. To utilize the MP translation facility, one inserts the following line in the main

program of the user's application code, as well as in any subprogram that performs MP operations:

```
USE MPMODULE
```

This line must be placed after the `PROGRAM, SUBROUTINE, FUNCTION` or `MODULE` statement, but before any implicit or type statements. This `USE` statement connects the subprogram with the Fortran-90 translation modules that define the MP datatypes and operator extensions.

At the beginning of the executable portion of the user's main program, even if the main program itself performs no MP operations, one inserts the line

```
CALL MPINIT
```

The routine `MPINIT` sets MPFUN library parameters, such as the precision level, and precomputes constants needed in transcendental function routines (see section four).

Three derived types are defined in the translation modules: `MP_INTEGER, MP_REAL` and `MP_COMPLEX`. In an application program, one may explicitly specify MP variables using Fortran-90 type statements, such as

```
TYPE (MP_INTEGER) IA, IB, IC
TYPE (MP_REAL) A, B, C, D, E
TYPE (MP_COMPLEX) Z
```

Alternatively, one may implicitly declare variables to be of one of the three MP types by using an `IMPLICIT` statement, such as

```
IMPLICIT TYPE (MP_REAL) (A-H, O-Z)
```

MP constants are handled a bit differently than with TRANSMP. These are now specified as literal constants, i.e. `'1.23456789'`. One may directly assign a MP constant to a MP variable, but if a MP constant appears in an expression, it must be as the argument to `MPINT, MPREAL` or `MPCMPL`, depending on whether it to be treated as MP integer, MP real or MP complex. Examples:

```
IA = '33333333333333333333333333333'
A = '1.4142135623 7309504880 168872420 E-10'
B = MPREAL ('1.25') / N
Z = 2 * MPCMPL ('1.2345', '6.7890')
```

Note that without the quotes to indicate an MP constant, the integer constant in the first line would overflow, and the floating constant in the second line would not be converted with full MP accuracy.

Quotes are not really required in the third line, since 1.25 can be converted exactly with ordinary arithmetic. However, note that simply writing `B = 1.25 / N` would not give a

fully accurate result if, for example, N is an ordinary integer with the value 7 (although it would be fine if N is 8). This is because the division operation would be performed using ordinary single precision arithmetic, and the inaccurate result would then be converted to MP and stored in B. The usage of the function MPREAL in the third line insures that the division is performed with MP arithmetic. This is an example of the care one must exercise in programming to insure that intermediate calculations are performed with MP arithmetic when required. In this respect, the new Fortran-90 translation system is like the FAST option of the TRANSMP program.

The expressions in lines three and four are examples of mixed mode operations. Virtually all such operations are allowed, and the result is of the type that one would expect. For example, the product of a MP real variable and an integer constant is of type MP real, and the sum of a complex variable and a MP real variable is of type MP complex. The only combinations that are not currently allowed are some exponentiations involving MP complex entities — these are defined only when the exponent is an integer.

Unformatted read and write statements with MP variables in the I/O list, such as WRITE (11) A, B, are handled as expected. But formatted and list-directed read and write statements, i.e. WRITE (6, *) A, B, will not produce the expected results for MP variables. These operations must now be handled using the special subroutines MPREAD and MPWRITE. The first argument of either routine is the unit number. Arguments 2-10 are the list of MP variables to be input or output. Within a single call to either routine, the MP variables in the list must all be the same type, either MP integer, MP real or MP complex. Examples:

```
CALL MPREAD (5, IA)
CALL MPWRITE (6, A, B, C, D, E)
```

An example of the format for input or output of MP numbers is

```
10 ^ 40 x -3.14159265358979323846264338327950288419 71,
```

On input, the exponent field is optional, and blanks may appear anywhere, but a comma must appear at the end of the last line of mantissa digits.

By default, only the first 56 mantissa digits of a MP number are output by MPWRITE, so that the output is contained on a single line. This output precision level can be changed by the user, either as a default setting or dynamically during execution (see section four).

It should be noted that the Fortran-90 translation modules generate calls to the standard arithmetic routines of the MPFUN library. If one wishes to utilize the "advanced" routines, which are intended for precision levels above 1000 digits (see section five), contact the author.

### 3. Multiprecision Functions

The functions MPINT, MPREAL and MPCMPL were mentioned in the previous section in the context of MP constants. These three functions are actually defined for all numeric

argument types, ordinary and MP. The result is MP integer, MP real or MP complex, respectively, no matter what the type of the argument. Thus one may use `MPREAL (A)` to convert the ordinary floating point variable `A` to MP real.

The corresponding Fortran type conversion functions `INT, REAL, DBLE, CMPLX`, and `DCMPLX` have also been extended to accept MP arguments. The result, in accordance with Fortran language conventions, is of type default integer, real, double precision, complex and double complex, respectively. Note that `REAL (IA)`, where `IA` is MP integer, is not of type MP real — if that is required, then `MPREAL` should be used instead.

Many of the other common Fortran intrinsics have been extended to accept MP arguments, and they return true MP values as appropriate. A complete list of the Fortran intrinsic functions that have been extended to MP is given in Table 1. In this table, the abbreviations I, R, D, C, DC, MPI, MPR, MPC denote default integer, real, double precision, double complex, MP integer, MP real and MP complex, respectively.

Some additional MP functions and subroutines that users may find useful are demonstrated in the following examples. Here `N` is an ordinary integer variable, and `A, B` and `C` are MP real.

```
A = MPRANF ()
B = MPNRTF (A, N)
CALL MPCSSNF (A, B, C)
CALL MPCSSHF (A, B, C)
```

These calls which produce a pseudorandom number in $(0, 1)$, the `N`-th root of `A`, both the cos and sin of `A`, and both the cosh and sinh of `A`, respectively. The above call to `MPNRTF` is equivalent to, but significantly faster than, the expression `A ** (MPREAL (1) / N)`. This is because the latter expression requires log and exp calculations, whereas `MPNRTF` uses an efficient Newton iteration scheme. Similarly, the above call to `MPCSSNF` executes faster than `B = COS (A)` and `C = SIN (A)`, although the results are the same. A similar comment applies to `MPCSSHF`.

## 4. Global Variables

There are a number of Fortran-90 global variables defined in the MP translation modules and in the MPFUN package. These variables, which are listed in Table 2, can be accessed by any user subprogram that includes a `USE MPMODULE` statement. The entries in the column labeled "Dynam. change" indicates whether the values of these variables may be dynamically changed by the user during execution of the program.

The first three global variables listed in Table 2 are set by the user in `PARAMETER` statements at the beginning of the file containing the MP translation modules. `MPIPL` is the initial precision level, in digits, and is often the only parameter that needs to be changed. `MPIOU`, the initial output precision level, is ordinarily set to 56, although it may be set to as high as `MPIPL` if desired. The parameter `MPIEP`, the initial MP "epsilon" level, is typically set to `10 - MPIPL` or so.

| Function Name | Arg. 1 | Arg. 2 | Result | Function Name | Arg. 1 | Arg. 2 | Result |
|---|---|---|---|---|---|---|---|
| ABS | MPI | | MPI | INT | MPI | | I |
| | MPR | | MPR | | MPR | | I |
| | MPC | | MPR | | MPC | | I |
| ACOS | MPR | | MPR | LOG | MPR | | MPR |
| AIMAG | MPC | | MPR | | MPC | | MPC |
| AINT | MPR | | MPR | LOG10 | MPR | | MPR |
| ANINT | MPR | | MPR | MAX | MPI | MPI | MPI |
| ASIN | MPR | | MPR | | MPR | MPR | MPR |
| ATAN | MPR | | MPR | MIN | MPI | MPI | MPI |
| ATAN2 | MPR | MPR | MPR | | MPR | MPR | MPR |
| CMPLX | MPI | MPI | C | MOD | MPI | MPI | MPI |
| | MPR | MPR | C | | MPR | MPR | MPR |
| | MPC | | C | NINT | MPR | | MPI |
| CONJG | MPC | | MPC | REAL | MPI | | R |
| COS | MPR | | MPR | | MPR | | R |
| | MPC | | MPC | | MPC | | R |
| COSH | MPR | | MPR | SIGN | MPI | MPI | MPI |
| DBLE | MPI | | D | | MPR | MPR | MPR |
| | MPR | | D | SIN | MPR | | MPR |
| | MPC | | D | | MPC | | MPC |
| DCMPLX | MPI | MPI | DC | SINH | MPR | | MPR |
| | MPR | MPR | DC | SQRT | MPR | | MPR |
| | MPC | | DC | | MPC | | MPC |
| EXP | MPR | | MPR | TAN | MPR | | MPR |
| | MPC | | MPC | TANH | MPR | | MPR |

Table 1: MP Extensions of Fortran Intrinsic Functions

| Variable Name | Type | Dynam. Change | Initial Value | Description |
|---|---|---|---|---|
| MPIPL | Integer | No | User sets | Initial (and maximum) precision, in digits. |
| MPIOU | Integer | No | User sets | Initial output precision, in digits. |
| MPIEP | Integer | No | User sets | $\log_{10}$ of initial MP epsilon. |
| MPWDS | Integer | No | See text | Initial (and maximum) precision, in words. |
| MPOUD | Integer | Yes | MPIOU | Current output precision, in digits. |
| MPEPS | MP real | Yes | $10^{\texttt{MPIEP}}$ | Current MP epsilon value. |
| MPL02 | MP real | No | $\log_e 2$ | |
| MPL10 | MP real | No | $\log_e 10$ | |
| MPPIC | MP real | No | $\pi$ | |
| MPNW | Integer | Yes | MPWDS | Current precision level, in words. |
| MPIDB | Integer | Yes | 0 | MPFUN debug level. |
| MPLDB | Integer | Yes | 6 | Logical unit for debug output. |
| MPNDB | Integer | Yes | 22 | No. of words in debug output. |
| MPIER | Integer | Yes | 0 | MPFUN error indicator. |
| MPMCR | Integer | Yes | 7 | Cross-over point for advanced routines. |
| MPIRD | Integer | Yes | 1 | MPFUN rounding option. |
| MPKER | Integer | Yes | 0 | Array of error options. |

Table 2: Global Variables

The call to MPINIT at the start of the user's main program sets initial values for the next six variables in the list. The final eight global variables in Table 2 are used in the MPFUN package and assume the values as shown. See [4] for additional details on the definition and usage of these variables.

As noted in Table 2, MPNW is the current numeric precision level, measured in machine words. On IEEE and most other systems, the approximate corresponding number of digits is given by $(\texttt{MPNW} - 1) \times 24 \log_{10} 2$. If one wishes to perform the same computation with a variety of precision levels without recompiling the translation modules, or if one needs to dynamically change the working precision level during the course of a calculation, this may be done by directly modifying the parameter MPNW in the user program, as in

```
MPNW = 127
```

But be careful not to change MPNW to a value larger than MPWDS, the initial precision level in words; otherwise array overwrite errors will occur. MPWDS is computed from MPIPL, the user-defined initial precision level in digits, using the expression int$[\texttt{MPIPL}/(24 \log_{10} 2) + 1]$,

where int denotes greatest integer. Because of possible numerical differences, it is recommended that users reference the system's value of MPWDS, rather than attempt to recalculate this value using the above formula. On Cray vector systems, the constant $24 \log_{10} 2 = 7.22472 \cdots$ in the above discussion should be replaced by $22 \log_{10} 2 = 6.62266 \cdots$.

With regards to the MP epsilon MPEPS, quotes should be used when changing the value of this variable, as in

```
MPEPS = '1E-500'
```

The quotes here insure that the constant is converted with full multiple precision. Without quotes, the constant will not be accurately converted, and in fact a constant of such a small size would result in an underflow condition on IEEE arithmetic systems.

**5. The Fortran-90 MPFUN Package**

The new Fortran-90 translation modules, like the older TRANSMP program, generate calls to the MPFUN library, which contains all of the subroutines that perform MP operations. With the advent of Fortran-90, the MPFUN library has also been updated to use some of the advanced features of the this language. Among the changes in the new MPFUN package are the elimination of common blocks and the dynamic allocation of scratch space. Thus the user never needs to worry about "insufficient scratch space" error messages.

One important algorithmic improvement introduced in the Fortran-90 version of the MPFUN library is the utilization of a faster scheme for multiple precision division, due to David M. Smith. The gist of this scheme is that it is not necessary to normalize the individual machine words of the trial quotient at every step of the division process. Instead, the normalization operation may be performed only occasionally. See [8] for details. On an IBM RS6000/590 workstation, this new division routine is as much as four times faster than the previous routine. One of the author's applications runs nearly twice as fast as a result, although a savings of 10% is more typical since relatively few applications are divide intensive. This improvement only affects the standard division routine. The advanced division routine, which is used for precision levels about about 1000 digits, is not affected.

Another algorithmic change is the utilization of an improved fast Fourier transform (FFT) algorithm [2], which is used by the advanced MP multiplication routine of MPFUN. This new FFT algorithm, which is variously called the "factored" or "four-step" FFT, exhibits significantly improved performance on computers that employ cache memory systems.

That this new FFT scheme is significantly more efficient on modern RISC systems can be seen from Table 3, which compares the performance of the new Fortran-90 MPFUN with the author's previous Fortran-77 MPFUN. These timings were performed on an IBM RS6000/590 workstation and compare the run time required to compute the constant $\pi$ to the specified precision levels (excluding binary to decimal conversion). The numbers of digits shown in the second column correspond to $2^m$ numbers of words, which are convenient precision levels for the FFT-based multiplication routine. Note that the new

| $m$ | Prec. Level (Digits) | CPU Time Old MPFUN | CPU Time New MPFUN |
|---|---|---|---|
| 4 | 115 | 0.0039 | 0.0035 |
| 5 | 231 | 0.0077 | 0.0068 |
| 6 | 462 | 0.0183 | 0.0160 |
| 7 | 924 | 0.0494 | 0.0440 |
| 8 | 1849 | 0.1250 | 0.0840 |
| 9 | 3699 | 0.3090 | 0.2640 |
| 10 | 7398 | 0.6670 | 0.6150 |
| 11 | 14796 | 1.4610 | 1.3900 |
| 12 | 29592 | 3.2860 | 3.1600 |
| 13 | 59184 | 13.3900 | 7.3500 |
| 14 | 118369 | 55.1200 | 16.7700 |
| 15 | 236739 | 150.3900 | 37.0400 |
| 16 | 473479 | 393.6800 | 83.4100 |

Table 3: Time to Compute $\pi$ on an IBM RS6000/590 Workstation

MPFUN package is up to 4.8 times faster than the old on this computation, even though the FFT routine only constitutes part of the operations being performed.

A third algorithmic change introduced in the Fortran-90 MPFUN package is the substitution of the author's PSLQ integer-relation finding algorithm [6, 1] for the HJLS algorithm [7] that was used in subroutine `MPINRL` of the Fortran-77 MPFUN. The PSLQ algorithm does not exhibit the catastrophic numerical instabilities that are a characteristic of the HJLS algorithm. With PSLQ, integer relations can be reliably detected when the precision level is set to only slightly higher than that of the input data.

One addition to the Fortran-90 MPFUN package is a routine to perform binary to decimal string conversion for extra-high precision arguments. This routine, named MPOUTX, employs a divide-and-conquer scheme, which together with the extra-high precision multiplication and division routines, permits rapid conversion of MP numbers whose precision ranges from roughly 1000 digits to millions of digits. MPOUTX uses the same calling sequence as the existing routine MPOUTC, which suffices for more modest precision levels. Extra scratch space is required for MPOUTX, but this space is automatically allocated by the routine when required.

## 6. An Application of the Fortran-90 Multiprecision System

In April 1993, Enrico Au-Yeung, an undergraduate at the University of Waterloo, brought to the attention of the author's colleague Jonathan Borwein the curious fact that

$$\sum_{k=1}^{\infty} \left(1 + \frac{1}{2} + \cdots + \frac{1}{k}\right)^2 k^{-2} \quad = \quad 4.59987\cdots \approx \frac{17}{4}\zeta(4) = \frac{17\pi^4}{360}$$

based on a computation to 500,000 terms. Borwein's reaction was to compute the value of this constant to a higher level of precision in order to dispel this conjecture. Surprisingly, his computation to 30 digits affirmed it. The present author then computed this constant to 100 decimal digits, and the above equality was still affirmed.

Intrigued by this result, the author developed computer programs, using the software described in this paper, to compute sums of this sort to high accuracy and to test if the numerical values satisfy simple formulas involving basic mathematical constants. Numerous experimental identities of this sort have now been obtained, and some of these have subsequently been established by rigorous proof. See [1] for details.

# References


[1] D. H. Bailey, J. M. Borwein and R. Girgensohn, "Experimental Evaluation of Euler Sums," *Experimental Mathematics* vol. 3, no. 1 (1994), pg. 17-30.

[2] D. H. Bailey, "FFTs in External or Hierarchical Memory," *Journal of Supercomputing*, vol. 4, no. 1 (March 1990), p. 23–35.

[3] D. H. Bailey, "Multiprecision Translation and Execution of Fortran Programs," *ACM Transactions on Mathematical Software*, vol. 19, no. 3, Sept. 1993, p. 288–319.

[4] D. H. Bailey, "A Portable High Performance Multiprecision Package," NASA Ames RNR Technical Report RNR-90-022.

[5] R. P. Brent, "A Fortran Multiple Precision Arithmetic Package," *ACM Transactions on Mathematical Software*, vol. 4 (1978), p. 57–70.

[6] H. R. P. Ferguson and D. H. Bailey, "A Polynomial Time, Numerically Stable Integer Relation Algorithm," NASA Ames RNR Technical Report RNR-91-032.

[7] J. Hastad, B. Just, J. C. Lagarias, and C. Schnorr, "Polynomial Time Algorithms for Finding Integer Relations Among Real Numbers," *SIAM Journal on Computing*, vol. 18 (1988), p. 859–881.

[8] D. M. Smith, "A Multiple Precision Division Alogorithm," to appear in *Mathematics of Computation*.