

# **ACCUMATH**

## **Extended Math Package Programming Manual**

© Copyright 1996, Pick Systems. All rights reserved.  
Printed in the United States of America.

# Introduction

**AccuMath** is an extended precision arithmetic utility package designed for programmers requiring more precision and significance than is available with BASIC. The package is implemented as a set of Assembler and BASIC subroutines that are callable from a BASIC program. The subroutines perform all of the standard arithmetic functions normally available, with almost unlimited significance and precision.

**AccuMath** operates on strings of decimal digits using decimal arithmetic, producing exact results for most operations, hence the designation "string arithmetic". This is in contrast to floating point arithmetic, which has a limited number of significant digits in which to represent a value, yielding inexact results.

**AccuMath** provides Assembler subroutines which perform the basic arithmetic operations: add, subtract, multiply and divide, as well as compare and adjust precision. These subroutines may be called by a BASIC program by using the `OCONV ( )` function.

**AccuMath** provides a set of BASIC subroutines which implement all the standard arithmetic operations and functions available in BASIC: add, subtract, multiply, divide, compare, precision, SIN, COS, TAN, EXP, PWR, LN and SQRT. The BASIC subroutines call the Assembler routines (possibly many times) to perform the primary arithmetic operations.

**AccuMath** also supports string arithmetic functions. These functions are compatible with Ultimate's string arithmetic functions. These functions are implemented using a pre-compiler, which translates the string function notation into the appropriate `OCONV ( )` notation transparently.

# Discussion

Throughout this document, we use the word **precision** to mean the number of fractional digits represented in a number; that is the number of digits to the right of the decimal point.

The phrase **significant digits** is used to indicate how many digits used to represent a number are valid. This includes both integer (to the left of the decimal point) and fractional (to the right of the decimal point) digits.

**Implicit precision** is one of the operating modes of the AccuMath utility package. In this mode, the number of fractional digits of the operands supplied to an operation imply the precision that is to be used by the operation. The precision of the operand with the greatest precision is used as the precision for the operation.

When using the implicit precision mode, operands may be set to any precision using the "set precision" function. Once set, operations using this operand will return results with the same (or greater if another operand had a greater precision) precision. When using this mode, operations like divide (and functions like SIN), do not require the precision to be explicitly specified.

The "set precision" function is used to increase or decrease the precision of an operand. When the precision is increased, trailing zeros are added after the last fractional digit of the number. When the precision is decreased, the number is rounded to the required precision.

**Explicit precision** is the other operating mode of the AccuMath utility package. In this mode, the precision of the operation may need to be explicitly specified. If the precision is not explicitly specified, a default precision may be used. The default precision is global, and the standard default is 14 digits. The default may be modified by using menu option 5 from the AccuMath menu.

When using the explicit precision mode, all operations and functions accept an optional explicit precision. If the explicit precision is omitted, then either the default precision (divide, SIN, COS, TAN, LN, EXP, PWR, SQRT), or the actual precision resulting from the operation (add, subtract, multiply) will be used. If the explicit precision is specified, the result of the operation or function will be rounded to the specified precision.

The "set precision" function may be used to decrease the precision of an operand, rounding the number to the specified precision.

In order to avoid inaccurate results and avoid confusion, it is recommended that either the **implicit precision** or **explicit precision** mode be used, but not both.

Since all of the arithmetic operations and functions are performed in decimal on strings of decimal digits, there is no formal limit on the number of significant digits which may be utilized. Likewise, there is no formal limit on the precision to which a number can be represented. However, there is an actual limit, determined by system configuration, which limits the size of the strings which may be manipulated and stored in a BASIC program. Generally, a maximum string length of 32000 digits is considered to be the limit.

Although there is no formal limit on significant digits, since operations are performed in decimal, as operands increase in length, the time required to execute operations and functions increases as well. Operands containing several hundred or thousands of digits may require considerable execution time, especially when using the SIN, COS, TAN, LN, PWR, EXP and SQRT functions.

Note that only the primitive operations add, subtract, multiply, divide and compare are implemented using Assembler subroutines. All other functions are implemented in BASIC as subroutines which call the Assembler subroutines many times.

AccuMath may be used in a BASIC program in any of 3 ways. First, the AccuMath BASIC subroutines may be CALLED to perform the required arithmetic operations. The subroutines are available in both **implicit precision** and **explicit precision** versions.

Next, the AccuMath Assembler subroutines may be called by using the BASIC OCONV() function. The definitions for the user-exits are contained in an item which may be INCLUDED in a BASIC program.

Last, AccuMath includes a pre-compiler, which can be used to compile programs using string arithmetic functions: SADD(), SSUB(), SMUL(), SDIV() and SCMP(). These functions are compatible with Ultimate's string arithmetic functions, and the pre-compiler allows BASIC source code compatibility with Ultimate.

When using AccuMath within a BASIC program, certain problems arise when combining data derived using the AccuMath subroutines with other numeric data in the program. BASIC will not recognize long strings of digits as numeric (when the binary value of the decimal number exceeds the limits for a numeric BASIC variable). Also, strings of digits with more fractional digits than specified in the PRECISION statement are not recognized as numeric. In order to minimize this problem, use the AccuMath "set precision" function to set the precision of the data to a BASIC compatible precision before using the data in standard BASIC calculations or storing the data in a file.

Also, large values derived using the AccuMath subroutines, and stored in files, may not be usable by the ACCESS/ENGLISH/RECALL function processor.

The following sections explain each of the methods of utilizing AccuMath in more detail.

# BASIC Subroutines

The BASIC subroutines are the recommended method of using AccuMath in your applications. They isolate the programmer from calling the Assembler subroutines (user-exits), and provide the greatest degree of portability and functionality.

The BASIC subroutines include all of the functions supported by BASIC. These are +, -, \*, /, ABS, COS, EXP, INT, LN, PWR, SIN, SQRT, and TAN. Two other functions that are not found in BASIC are CMP, used to compare two high precision numbers, and PRC, used to set the precision of an argument.

When using the BASIC subroutines, it is important to remember to use distinct variables (or expressions or constants) for each of the arguments passed or returned by the subroutine. Many implementations of BASIC give incorrect results when a variable is used for more than one argument in a subroutine. ***Never*** code a subroutine call like this:

```
CALL XADD(TOTAL,AMOUNT,TOTAL) incorrect usage!
```

In the above example, the variable TOTAL is passed to the subroutine in two places in the argument list. Instead, code the call like this:

```
TEMP=TOTAL  
CALL XADD(TEMP,AMOUNT,TOTAL)
```

## Addition

Two arguments may be added together using the XADD or XADDX subroutine. For XADD (or XADDX when PRECISION is null), the precision of the result will be the greater of the precision of either of the arguments. For XADDX with PRECISION not null, the result will be rounded to the specified precision.

```
CALL XADD(ARG1,ARG2,RESULT)  
  
CALL XADDX(ARG1,ARG2,PRECISION,RESULT)
```

## **Subtraction**

The difference between two arguments ( $ARG1 - ARG2$ ) may be computed using the XSUB or XSUBX subroutine. For XSUB (or XSUBX when PRECISION is null), the precision of the result will be the greater of the precision of either of the arguments. For XSUBX with PRECISION not null, the result will be rounded to the specified precision.

```
CALL XSUB ( ARG1 , ARG2 , RESULT )
```

```
CALL XSUBX ( ARG1 , ARG2 , PRECISION , RESULT )
```

## **Multiplication**

The product of two arguments may be computed using the XMUL or XMULX subroutine. For XMUL, the precision of the result will be the greater of the precision of either of the arguments; that is, the product will be rounded to the precision of the source argument with the greatest precision. For example, if  $ARG1=2.494$  (precision=3) and  $ARG2=1.23$  (precision=2) then  $RESULT=3.068$  (precision=3) which is 3.06762 rounded to precision 3. For XMULX with PRECISION is null, the precision of the result will be sum of the precision of each of the arguments. For example, if  $ARG1=1.25$  (precision=2) and  $ARG2=0.375$  (precision=3), then  $RESULT=0.46875$  (precision=5). For XMULX when PRECISION not null, the product will be rounded to the specified precision.

```
CALL XMUL ( ARG1 , ARG2 , RESULT )
```

```
CALL XMULX ( ARG1 , ARG2 , PRECISION , RESULT )
```

## **Division**

The quotient of two arguments ( $ARG1 / ARG2$ ) may be computed using the XDIV or XDIVX subroutine. For XDIV, the precision of the result will be the greater of the precision of either of the arguments. The least significant digit of the quotient will be rounded correctly. For XDIVX when PRECISION is not null, the result will be rounded to the specified precision. For XDIVX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

```
CALL XDIV ( ARG1 , ARG2 , RESULT )
```

```
CALL XDIVX ( ARG1 , ARG2 , PRECISION , RESULT )
```

## **Absolute Value**

The XABS or XABSX subroutine returns the absolute numeric value of the argument. An absolute value is the numerical value of a number without reference to its algebraic sign. While the result looks positive it is actually unsigned. The precision of the result is the precision of argument. For XABSX, PRECISION is ignored.

```
CALL XABS ( ARG1 , RESULT )
```

```
CALL XABSX ( ARG1 , PRECISION , RESULT )
```

## **Cosine of an Angle**

The XCOS or XCOSX subroutine is used to generate the cosine of an angle expressed in degrees. For XCOS, the precision of the result is the precision of the argument. For XCOSX when PRECISION is not null, the result will be rounded to the specified precision. For XCOSX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

## **Exponential**

The XEXP or XEXPX subroutine may be used to raise the natural logarithm base,  $e$  (2.71828...), to a power. For XEXP, the precision of the result is the precision of the argument. For XEXPX when PRECISION is not null, the result will be rounded to the specified precision. For XEXPX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

```
CALL XEXP ( ARG1 , RESULT )
```

```
CALL XEXPX ( ARG1 , PRECISION , RESULT )
```

## **Integer Numeric Value**

The XINT or XINTX subroutine is used to return the integer portion of the specified expression. The fractional portion of the argument is truncated. For XINTX, PRECISION is ignored.

```
CALL XINT( ARG1 , RESULT )  
  
CALL XINTX( ARG1 , ARG2 , RESULT )
```

## **Natural Logarithm**

The XLN or XLNX subroutine generates the natural (base  $e$ ) logarithm of the argument. The LN function is the inverse of the EXP function. For XLN, the precision of the result is the precision of the argument. For XLNX when PRECISION is not null, the result will be rounded to the specified precision. For XLNX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

```
CALL XLN( ARG1 , RESULT )  
  
CALL XLNX( ARG1 , PRECISION , RESULT )
```

## **Raising by a Power**

The XPWR or XPWRX subroutine raises the first argument to the power specified in the second argument. If the second argument is zero, the function will return the value one. For XPWR, the precision of the result will be the greater of the precision of either of the arguments. For XPWRX when PRECISION is not null, the result will be rounded to the specified precision. For XPWRX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

```
CALL XPWR( ARG1 , ARG2 , RESULT )  
  
CALL XPWRX( ARG1 , ARG2 , PRECISION , RESULT )
```

## **Sine of an Angle**

The XSIN or XSINX subroutine is used to generate the sine of an angle expressed in degrees. For XSIN, the precision of the result is the precision of the argument. For XSINX when PRECISION is not null, the result will be rounded to the specified precision. For XSINX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

```
CALL XSIN(ARG1,RESULT)
```

```
CALL XSINX(ARG1,PRECISION,RESULT)
```

## **Square Root**

The XSQRT or XSQRTX subroutine will return the positive square root of any positive number. For XSQRT, the precision of the result is the precision of the argument. For XSQRTX when PRECISION is not null, the result will be rounded to the specified precision. For XSQRTX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

```
CALL XSQRT(ARG1,RESULT)
```

```
CALL XSQRTX(ARG1,PRECISION,  
RESULT)
```

## **Tangent of an Angle**

The XTAN or XTANX subroutine is used to generate the tangent of an angle expressed in degrees. For XTAN, the precision of the result is the precision of the argument. For XTANX when PRECISION is not null, the result will be rounded to the specified precision. For XTANX when PRECISION is null, the precision of the result will be the default precision (standard default is 14).

The result will contain no more than 48 significant digits.

```
CALL XTAN(ARG1,RESULT)
```

```
CALL XTANX(ARG1,PRECISION,RESULT)
```

## Comparison

Two arguments may be compared, and the results (lower, equal, higher) determined. If the first argument is less than the second argument then -1 is returned. If the first argument is greater than the second argument then 1 is returned. If the first argument is equal to the second argument then 0 is returned. For XCMPX, PRECISION is ignored.

```
CALL XCMP ( ARG1 , ARG2 , RESULT )
```

```
CALL XCMPX ( ARG1 , ARG2 , PRECISION , RESULT )
```

## Precision

The precision of a value may be set using the XPRC or XPRCX subroutine. The first argument contains the value whose precision is being changed, and the second argument contains the desired precision. The precision must be a non-negative integer less than 32000.

If the precision of the first argument is greater than the second argument, then the first argument is rounded to precision specified by the second argument. If the precision of the first argument is less than the second argument, then trailing zeros (and possibly a decimal point) are added to the end of the first argument forcing it to precision specified by the second argument.

XPRC and XPRCX are identical. Both routines are included only for symmetry.

```
CALL XPRC ( ARG1 , ARG2 , RESULT )
```

```
CALL XPRCX ( ARG1 , ARG2 , RESULT )
```

# Assembler Subroutines

The Assembler subroutines are another method of using AccuMath in your applications. The main advantage of using the Assembler subroutines is that they are faster than the BASIC subroutines, mainly due to the overhead of the BASIC CALL statement. The main disadvantages of using the Assembler subroutines are that they require the programmer to call the Assembler subroutines using user-exits, and they only provide the primary arithmetic operations: add, subtract, multiply, divide, compare and precision. If execution speed is a priority however, the Assembler subroutines are an acceptable method.

All of the Assembler subroutines (except XPRC) have two modes, implicit precision and explicit precision. Their operation is exactly as described in the preceding section (BASIC subroutines).

When calling the Assembler subroutines, the following statement should be included near the beginning of the program:

```
INCLUDE XP XPA.DEFS
```

Each of the Assembler subroutines is defined in the XPA.DEFS item, and may then be referred to by a symbolic name.

The Assembler subroutines are called by using the BASIC OCONV function. The arguments are concatenated together (using attribute marks to separate them), and passed as data to be converted. The AccuMath function is used as the conversion code.

The following sections describe the Assembler subroutines in more detail.

## Addition

Two arguments may be added together using the XADD or XADDX subroutine. For XADD (or XADDX when PRECISION is omitted), the precision of the result will be the greater of the precision of either of the arguments. For XADDX when PRECISION not null, the result will be rounded to the specified precision.

```
RESULT = OCONV ( ARG1 : AM : ARG2 , XADD )
```

```
RESULT = OCONV ( ARG1 : AM : ARG2 : AM : PRECISION , XADDX )
```

## Subtraction

The difference between two arguments (ARG1 - ARG2) may be computed using the XSUB or XSUBX subroutine. For XSUB (or XSUBX when PRECISION is omitted), the precision of the result will be the greater of the precision of either of the arguments. For XSUBX when PRECISION not null, the result will be rounded to the specified precision.

```
RESULT = OCONV ( ARG1 : AM : ARG2 , XSUB )
```

```
RESULT = OCONV ( ARG1 : AM : ARG2 : AM : PRECISION , XSUBX )
```

## Multiplication

The product of two arguments may be computed using the XMUL or XMULX subroutine. For XMUL, the precision of the result will be the greater of the precision of either of the arguments; that is, the product will be rounded to the precision of the source argument with the greatest precision. For example, if ARG1=2.494 (precision=3) and ARG2=1.23 (precision=2) then RESULT=3.068 (precision=3) which is 3.06762 rounded to precision 3. For XMULX when PRECISION is omitted, the precision of the result will be sum of the precision of each of the arguments. For example, if ARG1=1.25 (precision=2) and ARG2=0.375 (precision=3), then RESULT=0.46875 (precision=5). For XMULX when PRECISION not null, the product will be rounded to the specified precision.

```
RESULT = OCONV ( ARG1 : AM : ARG2 , XMUL )
```

```
RESULT = OCONV ( ARG1 : AM : ARG2 : AM : PRECISION , XMULX )
```

## Division

The quotient of two arguments (ARG1 / ARG2) may be computed using the XDIV or XDIVX subroutine. For XDIV, the precision of the result will be the greater of the precision of either of the arguments. The least significant digit of the quotient will be rounded correctly. For XDIVX when PRECISION is not null, the result will be rounded to the specified precision. For XDIVX when PRECISION is omitted, the precision of the result will be the default precision (standard default is 14).

```
RESULT = OCONV ( ARG1 : AM : ARG2 , XDIV )
```

```
RESULT = OCONV ( ARG1 : AM : ARG2 : AM : PRECISION , XDIVX )
```

## Comparison

Two arguments may be compared, and the results (lower, equal, higher) determined. If the first argument is less than the second argument then -1 is returned. If the first argument is greater than the second argument then 1 is returned. If the first argument is equal to the second argument then 0 is returned. For XCMPX, PRECISION is ignored.

```
RESULT = OCONV ( ARG1 : AM : ARG2 , XCMP )
```

```
RESULT = OCONV ( ARG1 : AM : ARG2 : AM : PRECISION , XCMPX )
```

## Precision

The precision of a value may be set using the XPRC subroutine. The first argument contains the value whose precision is being changed, and the second argument contains the desired precision. The precision must be a non-negative integer less than 32000.

If the precision of the first argument is greater than the second argument, then the first argument is rounded to precision specified by the second argument. If the precision of the first argument is less than the second argument, then trailing zeros (and possibly a decimal point) are added to the end of the first argument forcing it to precision specified by the second argument.

```
RESULT = OCONV ( ARG1 : AM : ARG2 , XPRC )
```

# String Arithmetic Functions

The string arithmetic functions and pre-compiler are the last method of using AccuMath in your applications. These functions are compatible with the Ultimate string math functions. Using the pre-compiler, source code compatibility with programs using Ultimate's string arithmetic functions can be maintained.

Only SADD(), SSUB(), SMUL(), SDIV() and SCMP() are defined as string arithmetic functions. These functions will accept either two or three arguments. The first two arguments are the two operands supplied to the function. The third, optional operand, is an explicit precision for the function. These functions behave exactly like the explicit precision subroutines (if the third argument is missing, the default precision is used).

If the other AccuMath functions are required, call the explicit precision mode BASIC subroutines.

For compatibility with Ultimate string arithmetic, use only two arguments in the string arithmetic functions. The third argument, allowing the specification of function precision, is an extension added to AccuMath.

## Pre-Compiler

When using the string arithmetic functions, it is necessary to compile your programs using the AccuMath pre-compiler: XP-BASIC. The pre-compiler translates the string arithmetic functions into OCONV() functions. It then calls the BASIC compiler to compile the program.

When using the pre-compiler, it is necessary to include the following statement in your program:

```
INCLUDE XP XPA.DEFS
```

If a program INCLUDEs a program fragment which contains string arithmetic functions, they will not be converted by the pre-compiler, and will not compile properly. In this case, the pre-compiler may be used to translate the INCLUDEd program fragment and update the source code with the translated version. To do this, copy the original source program fragment to another file (since the source code will be updated). Next, execute the XP-PRECOMP verb with the (O) option to update the source code. Now when you compile the main program, the INCLUDEd program fragment will have been translated by XP-PRECOMP, and the program will compile successfully.

The syntax for the XP-BASIC and XP-PRECOMP verbs is exactly the same as the BASIC verb.

Note: for Open Architecture only, XP-BASIC and XP-PREBASIC are case insensitive, and XP-COMPILE and XP-PRECOMP are case sensitive.

The following sections describe the string arithmetic functions in more detail.

### **Addition**

Two arguments may be added together using the SADD function. If the third argument is omitted, the precision of the result will be the greater of the precision of either of the arguments. If the third argument is specified, the result will be rounded to the specified precision.

```
RESULT = SADD( ARG1 , ARG2 )
```

```
RESULT = SADD( ARG1 , ARG2 , PRECISION )
```

### **Subtraction**

The difference between two arguments (ARG1 - ARG2) may be computed using the SSUB function. If the third argument is omitted, the precision of the result will be the greater of the precision of either of the arguments. If the third argument is specified, the result will be rounded to the specified precision.

```
RESULT = SSUB( ARG1 , ARG2 )
```

```
RESULT = SSUB( ARG1 , ARG2 , PRECISION )
```

## **Multiplication**

The product of two arguments may be computed using the SMUL function. If the third argument is omitted, the precision of the result will be sum of the precision of each of the arguments. For example, if ARG1=1.25 (precision=2) and ARG2=0.375 (precision=3), then RESULT=0.46875 (precision=5). If the third argument is specified, the product will be rounded to the specified precision.

```
RESULT = SMUL ( ARG1 , ARG2 )
```

```
RESULT = SMUL ( ARG1 , ARG2 , PRECISION )
```

## **Division**

The quotient of two arguments (ARG1 / ARG2) may be computed using the SDIV function. If the third argument is omitted, the precision of the result will be the default precision (standard default is 14). If the third argument is specified, the result will be rounded to the specified precision.

```
RESULT = SDIV ( ARG1 , ARG2 )
```

```
RESULT = SDIV ( ARG1 , ARG2 , PRECISION )
```

## **Comparison**

Two arguments may be compared, and the results (lower, equal, higher) determined. If the first argument is less than the second argument then -1 is returned. If the first argument is greater than the second argument then 1 is returned. If the first argument is equal to the second argument then 0 is returned. A third argument, if specified, is ignored.

```
RESULT = SCMP ( ARG1 , ARG2 )
```

```
RESULT = SCMP ( ARG1 , ARG2 , PRECISION )
```

# Calculator

The AccuMath package contains two algebraic calculator programs. These may be used to aid in developing programs utilizing the AccuMath subroutines, testing formulas, and for general use. One of the calculators (ALG) uses the implicit precision mode, the other (ALGX) uses the explicit precision mode.

To use the calculator program, type **ALG** or **ALGX** at TCL. If you run ALGX, the program will prompt:

```
Maximum precision for calculations (RETURN for default)?
```

Enter any valid precision (4-50), or press RETURN to use the default precision (standard default precision is 14).

Next, either ALG or ALGX will prompt:

```
Enter Expression:
```

At this prompt, you can enter any valid expression in standard algebraic notation. The syntax is similar to BASIC, except that the only operands allowed are numbers or references to the calculator's memory elements.

If a valid expression was entered, the calculator will print the result:

```
1> result
```

After the result is printed, the calculator will again prompt for an expression.

As each result is displayed, a memory element number is printed before it. The result may be used as an operand in another expression by preceding the memory element number by a percent sign. For example, consider the following sequence:

```
Enter Expression: SIN(23)
1> 0.39073112848927

Enter Expression: COS(23)
2> 0.92050485345244

Enter Expression: (%1)^2 + (%2)^2
3> 1

Enter Expression: 2*(1.34563421349784/.20145265)^3
4> 596.06291722886108

Enter Expression: 100000^1.1875/(5*.140328642283449)
5> 1234194.68651440253212

Enter Expression: LN(1.12)
6> 0.113328685307

Enter Expression: 500000/(1+0.0675)^5
7> 360687.07992475241642
```

# Examples

## Example 1 - Future Value

This example computes future value based on present value, interest rate and time. The example uses the implicit precision mode BASIC subroutines. The entry, compilation and running of the program is shown below.

```
>ED BP FV
Top
001 * Calculate future value
002 PRINT 'Enter present value : ';; INPUT PV
003 PRINT 'Enter number of years : ';; INPUT T
004 PRINT 'Enter annual percentage rate : ';; INPUT I
005 * Calculate:  $FV = PV \cdot (1 + (I/100))^T$ 
006 CALL XDIV(I, '100.000000000000', W) ;* sets precision to 12
007 CALL XADD(W, 1, X) ;*  $1 + (I/100)$ 
008 CALL XPWR(X, T, Y) ;*  $(1 + (I/100))^T$ 
009 CALL XMUL(PV, Y, Z) ;*  $PV \cdot (1 + (I/100))^T$ 
010 CALL XPRC(Z, 2, FV) ;* round to 2 places
011 PRINT 'Future value = ':FV
012 END
.FI
'FV' filed.

>BASIC BP FV
*****

>RUN BP FV
Enter present value : ?10000
Enter number of years : ?5
Enter annual percentage rate : ?7.15102567111
Future value = 14124.78
```

## Example 2 - Loan Amortization

This example uses the string math functions and the explicit precision mode (the string math functions by default use the explicit precision mode, and the BASIC subroutine XPWRX is the explicit precision version). Although the explicit precision mode is used, the precision is never explicitly specified, thus using the default precision (standard default precision is 14). In this example, the pre-compiler XP-BASIC is used to compile the program, rather than BASIC. The pre-compiler translates the string math functions into the appropriate OCONV( ) functions, and then enters the normal BASIC compiler.

```
>ED BP AMORT
Top
001 * Loan Amortization
002 INCLUDE XP XPA.DEFS
003 PRINT 'Enter principal: ':: INPUT PRINCIPAL
004 PRINT 'Enter months  : ':: INPUT PERIODS
005 PRINT 'Enter interest : ':: INPUT RATE
006 RATE = SDIV(RATE,1200)
007 CALL XPWRX(SADD(1,RATE),PERIODS,'',X)
008 PAYMENT = SMUL(PRINCIPAL,SDIV(RATE,SSUB(1,SDIV(1,X))))
009 PRINT 'Monthly payment = ':PAYMENT
010 END
.FI
'AMORT' filed.

>XP-BASIC BP AMORT
*****

>RUN BP AMORT
Enter principal: ?250000
Enter months   : ?360
Enter interest : ?9.5
Monthly payment = 2102.1355179475
```