Uniform Numeric Strings

New Subroutines: cv_fixed_point_string_, radix_indicator_string_, cv_condition_

Author: Gary Dixon
Date: January 3, 2022

Abstract

MCR10099: Enhancements to index_set Command/AF points out many inconsistencies in the use of numeric strings by Multics commands, active functions and subroutines. This bulletin describes the full nature of the problem, outlines an approach for solving the problem, and proposes several new subroutines to begin a solution. Changes in several command/active function programs will make use of the new subroutines. Future MTBs and MCRs will further implementation of the plan.

Table 1: Revision History

Date	Revision	Author	Comment
2022-01-03	1.0	Gary Dixon	Initial draft.

Table of Contents

List of Tables

Introduction

MCR10099 (https://s3.amazonaws.com/eswenson-multics/public/mcrs/MCR10099_index_set_v1.0.pdf) installs a new version of the index_set command/AF that outputs sequences of integers in various numeric bases. Numbers output in non-decimal bases include a radix indicator sub-field which specifies the numeric base. But many Multics programs do not accept such non-decimal numbers as input.

A *radix indicator* is a sub-field appended to the string representation of a number to indicate the numeric base (or *radix*) of the digits expressing the numeric value. This indicator is often a single letter: o for octal numbers, x for hexadecimal or base 16 numbers, etc. The letter can be preceded by an underscore to disambiguate the final digit of a hexadecimal number (like b and d) from its possible use as a radix indicator (b for binary, d for decimal number). Without the leading underscore, it is a hex digit; with the underscore, it is a radix indicator.

The complete list of radix indicators in use today is shown below. No program supports all of these indicators. Several programs support some of them. This inconsistency is part of the overall problem.

```
List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).
rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

Other aspects of the problem are summarized below.

- Several command/active functions that handle integer strings (e.g., decimal, octal, hexadecimal, binary) accept some of these radix indicators: b, q, o and x but NOT d or rN.
- The related subroutines (cv_dec_, cv_oct_, cv_hex_ and cv_binary_) are called by many Multics and user programs to convert numeric strings, but these programs do not accept any radix indicators.
- The cv_integer_string_ subroutine supports some of them: b, d, o, x and rN but NOT q.
- Most programming languages (PL/I, FORTRAN, COBOL, etc.) define rules for converting between character string representations of numbers and target data types. Such rules do not support the addition of a radix indicator to string representations.

These differences in accepted formats can lead to user frustration, with numbers output by one program not being accepted as input to another program. Therefore, a new Multics-wide goal is needed.

Goal: All Multics commands, active functions, and subroutines that accept numeric strings should follow the same formatting rules for the string representations of integer and fixed-point values.

Such goal may be difficult to achieve. Changing programming language standards is outside the scope of a Multics-wide goal. Such goal must be limited to Multics commands, active functions and subroutines.

The approach proposed here includes the following steps:

A. Centralize handling of the radix indicator sub-field in a new subroutine: radix_indicator_string_.pl1.

B. Centralize handling of fixed-point string conversion in a new subroutine: cv_fixed_point_string_.pl1.

C. Change known command/AFs performing float dec(59) real number conversions which now include (or should accept) radix indicators to call cv_fixed_point_string_.

D. Centralize handling of integer string conversion in the existing subroutine: cv_integer_string_.pl1.

E. Merge related subroutines performing an integer string conversion into cv_integer_string_.pl1.

F. Change known command/AFs performing integer string conversions to a fixed bin(35) data type to instead call cv_integer_string_.

G. Watch for other commands, active functions, subsystem requests, and subroutines converting an integer string to an arithmetic value. When found in the future, propose a change to have the program use one of the already-changed subroutines to do its conversions.

This MTB expands on steps A, B and C by proposing the following changes.

1. Create a new radix_indicator_string_.pl1 subroutine to centralize processing of a radix indicator sub-field appended to the end of any numeric string. The indicator begins with an optional underscore, followed by one or more indicator characters. This subroutine accepts the superset of indicator characters currently supported in various commands: letters b q o d x; plus the more general rN radix string covering bases from 2 through 16. Indicator characters may be either uppercase or lowercase. radix_indicator_string_ will be installed with cv_integer_string_ in the bound_library_1_ object segment.
2. Create a new cv_fixed_point_string_.pl1 which converts strings to the float decimal(59) data type. Such *fixed-point strings* may include a radix point in their mantissa, followed by optional exponent sub-field and a radix indicator sub-field. This subroutine will accept numbers expressed in bases 2 through 16, where either the default_base argument or trailing radix indicator specify that numeric base. It will call radix_indicator_string_ to evaluate any radix indicator sub-field.

3. Change programs that operate with float decimal(59) values to call cv_fixed_point_string_. This MTB includes changes for the following command/AFs.

   hex.pl1: binary octal decimal hexadecimal
   plus.pl1: plus minus times divide quotient mod max min trunk floor ceil

4. The calc.pl1 command/AF currently performs arithmetic using the float bin(27) data type. Change it to use the more precise float decimal(59) data instead, and to call cv_fixed_point_string_ to convert numeric strings to that data type. It will call numeric_to_ascii_ to convert its float decimal(59) numbers to output strings. This change would eliminate the separate ffip.pl1 and ffop.pl1 support routines currently called by calc for such conversions, leaving only calc.pl1 in bound_calc_. The revised calc.pl1 will therefore be moved into bound_full_cp_ where hex.pl1 now resides.

5. Create a new cv_condition_.pl1 subroutine to simplify reporting of conditions that can occur when converting numeric strings to an arithmetic data type: conversion, fixedoverflow, overflow, size, and underflow. It will provide PL/I oncode information to characterize each condition, plus onsource, onchar, and onchar_index details for a conversion condition. Much of this information is difficult to obtain when establishing an on-unit in a program accepting numeric inputs.

6. The interpret_info_struc_.pl1 subroutine is called by condition_interpreter_, default_error_handler_, et al. to create an error message describing arithmetic conditions like conversion, overflow, size, and underflow. Change it to provide more detailed information about a conversion condition: which digit position, exponent character, or radix indicator character caused the condition.

Future MTBs will cover other parts of the plan.

New Subroutine: radix_indicator_string_.pl1

Several programs accept a subset of radix indicators: binary, octal, decimal, and hex command/AFs; cv_integer_string_ subroutine. Each includes their own code which locates and identifies the radix indicator sub-field, sets the base for interpreting the digits in the string, and reports errors found in the radix indicator sub-field of the numeric string.

Because the radix indicator appears at the end of a numeric string but must be evaluated before any digits in the string are converted, non-trivial coding is required. This code has several nuances not immediately obvious to a developer. None of the existing code correctly implements the full set of indicator characters and the general rN format.

The best approach is to centralize handling of the radix indicator in a new subroutine: radix_indicator_string_.pl1. Given full content of an input numeric string and the default_base expected for that string, the subroutine will:

- Determine if the string ends with any form of radix indicator.
- If not, the subroutine:
  - Returns to caller with chosen_base set to default_base.
- If radix indicator sub-field is found, the subroutine:
  - Determines the length of the radix sub-field.
  - Evaluates that sub-field to determine the base of the numeric string.
  - Returns to caller:
    - Length of the numeric string with radix indicator sub-field removed.
    - Chosen base specified by any radix indicator; or the default_base.
    - Character position of first invalid character in the radix sub-field.
    - Status code indicating either an invalid default_base value; or conversion error in evaluating the radix indicator sub-field of the numeric string.

The interface proposed for the radix_indicator_string_ subroutine is shown in its proposed info segment:

```
:Info: radix_indicator_string_:  2021-11-29  radix_indicator_string_

Function: This subroutine checks a numeric string to determine whether
it ends with a radix indicator.  If such indicator is appended to the
string, it specifies a numeric base (or radix) used to express
digits in the string.  See "List of radix indicators" for information
about the formats of a radix indicator string.

Syntax:
declare radix_indicator_string_ entry (char(*), fixed bin, fixed bin,
   fixed bin, fixed bin, fixed bin(35));
call radix_indicator_string_( number_string, default_base,
   number_length_without_radix, effective_base, radix_fail_index,
   code);

Arguments:
number_string
   is a numeric string which may end with a radix indicator.
   (Input)
default_base
   is the base to be used if the input string does not have a trailing
   radix indicator.  This base may be any integer between 2 to 16
   inclusive. (Input)
number_length_without_radix
   is the length of the portion of number_string with any trailing
   whitespace and any radix indicator removed.  The calling program
   must interpret these leading characters to complete conversion of
   number_string to an arithmetic data type. (Output)

effective_base
   is either default_base, or the base specified by any radix
   indicator string appended to the number_string.  The calling
   program should interpret remaining characters of number_string
   using this numeric base. (Output)
radix_fail_index
   index within number_string of the leftmost incorrect character
   of a radix indicator. (Output)
code
   is one of the following status code if an error was found when
   examining the radix indicator. (Output)
    error_table_$bad_arg:         unsupported default_base value.
    error_table_$bad_conversion:  invalid radix indicator.

List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
```

```
   between 2 and 16 inclusive.

Notes:  The following subroutines are known to accept numbers which
include optional radix indicators:
    cv_integer_string_, cv_integer_string_check_
    cv_binary_, cv_binary_check_
    cv_oct_, cv_oct_check_
    cv_dec_, cv_dec_check_
    cv_hex_, cv_hex_check_


The following command/active functions are known to accept numbers
which include optional radix indicators:
    binary, bin
    octal, oct
    decimal, dec
    hexadecimal, hex

    plus, minus, times, divide, quotient, mod, max, min, trunk, floor,
      ceil, round
    calc


Use a radix indicator beginning with underscore (_) if the
effective numeric base includes "b" and "d" as valid digits.

For example, if the effective numeric base is 16, a hexadecimal string
might end with a "d" digit (e.g., 10d) in which the final "d" is
treated as a digit in the number.  To treat it as a radix indicator,
use a radix indicator beginning with underscore (e.g., 10_d).


Examples:
The following number strings show uses of a radix indicator.  These
examples assume decimal as the default numeric base.

Hexadecimal numbers:  100x              100.01x
Decimal numbers:      256   256d  256.  2.56E+2_d
Octal numbers:        400o  400.o       400.002o
Binary numbers:       100000000b        100000000.00000001b
```

The radix_indicator_string_ subroutine will be added to bound_library_1_, which also contains the cv_integer_string_ subroutine, and will contain the cv_fixed_point_string_ subroutine. Both of these cv_XXX_ routines are primary callers of radix_indicator_string_. The new radix_indicator_string_ must co-reside with them in bound_library_1_ because these cv_XXX_ subroutines are used by BCE software, and by active functions available in the BCE that call one of these cv_XXX_ routines.

The bound_library_1_.bind file will add the radix_indicator_string_ name to the bound object, and retain that entry point in the radix_indicator_string_ object.

New Subroutine: cv_fixed_point_string_.pl1

Several programs accept fixed-point numeric strings (e.g., -123.45) and use the float dec(59) data type for their arithmetic target. plus.pl1 is a prime example. Others use float bin(27) data type. calc.pl1 is an example.

float decimal(59) supports: the greatest precision (up to 59 decimal digits); and the broadest range of numeric values with exponents from 10^{-128} to 10^{+127}. Therefore, this data type was chosen for implementation of the new cv_fixed_point_string_.pl1 subroutine.

An attempt was made to add an entry point which converted the input string to float dec(59) and then converted that value to float bin(27). However, this approach does not provide the same level of accuracy in the decimal-to-binary conversion as doing a conversion directly from input string to float bin(27). Furthermore, calc.pl1 is the most significant program using float bin(27). It was coded before the Extended Instruction Set (EIS) instructions of the Multics hardware added support for float decimal values.

It would be more beneficial to upgrade calc.pl1 to use float decimal(59) rather than trying to provide better accuracy in a cv_fixed_point_string_ entry point supporting the older float bin(27) conversions for non-decimal bases.

The new cv_fixed_point_string_.pl1 accepts an ASCII representation of a fixed-point real number and a default_base in which digits of that number are expressed. It converts the string into a float decimal(59) data type.

The fixed-point string representation is composed of one, two, or three sub-fields.

```
+123456789.0034E+02_d
\_ mantissa __/\_/ \/
               |  |
      exponent –'  '– radix indicator
```

A single sub-field string has just a *mantissa*.

- **mantissa**: an optionally signed sequence of 1 to N digits, plus an optional radix point (period or dot character). Digit values in numeric bases from 2 to 16 are accepted. The maximum precision N varies depending upon the effective base in which the string is expressed. See the table below. Example: -12.345

Table 2: Maximum Precision for Each Numeric Base (in digits)

BASE	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Precision N	199	125	99	85	77	70	66	62	59	57	55	53	52	51	49

Digits for bases 11 through 16 may be entered using letters for digit values beyond 9, with the letters accepted in either uppercase or lowercase. Digits for the most common numeric bases are shown in the table below, ordered in ascending value.

BASE	VALID DIGITS IN THAT BASE
2	0 1
8	0 1 2 3 4 5 6 7
10	0 1 2 3 4 5 6 7 8 9
12	0 1 2 3 5 5 6 7 8 9 A B
16	0 1 2 3 4 5 6 7 8 9 A B C D E F

For a fixed-point string expressed in a base in the range 2 through 13, the mantissa may be followed an *exponent* sub-field.

- **exponent**: begins with one of the exponent indicator letters: D E F followed by a decimal integer EXP in the range: -128 <= EXP <= 127

The various exponent indicator letters convey differing attributes when used in literal constants in a PL/I or FORTRAN source program: E for floating-point, D for double-precision floating-point, and F for fixed-point. When given in a fixed-point numeric string argument to cv_fixed_point_string_, their only meaning is to indicate the start of the exponent sub-field.

An exponent sub-field is not accepted in a string expressed in bases 14 through 16 because the exponent indicator letter would be treated as a valid digit in those bases.

A positive exponent EXP shifts the radix point to the right by EXP radix places; a negative EXP shifts the radix point left by EXP places. Use of an exponent in decimal strings is shown in the table below.

STRING	mantissa * BASE**EXP	RESULT
123E-4	123 * 10**-4	0.0123
1.23E+1	1.23 * 10**+1	12.3
1.23E5_d	1.23 * 10**+5	123000
.123E+2d	.123 * 10**+2	12.3

Any fixed-point string may end with a *radix indicator* sub-field.

- **radix indicator**: a letter or string appended to a numeric string which specifies the numeric base to be used in assigning values to digits in the string. Example using an octal radix: -12.345o

The radix_indicator_string_ subroutine is called by cv_fixed_point_string_ to look for a radix indicator. If one is present, its value overrides the default_base argument in specifying the effective base of the fixed-point string. Formats accepted for a radix indicator string are shown with examples in the table below.

INDICATOR	NUMERIC BASE	EXAMPLES	VALUE (base 10)
b or _b	base 2 or binary	10001.11b	17.75
q or _q	base 4 or quaternary	113_q	23
o or _o	base 8 or octal	21.6o	17.75
d or _d	base 10 or decimal	17.75d	17.75
x or _x	base 16 or hexadecimal	11.Cx	17.75
rN or _rN	N is a decimal base number: 2 <= N <= 16	43r5	23
		15.9_r12	17.75

Documentation for the new interface is shown below.

```
2021-12-26  cv_fixed_point_string_

Function: accepts an ASCII representation of a fixed-point number with
optional exponent and radix indicator sub-fields.  It returns the
float decimal(59) representation of that number.

Syntax:
declare cv_fixed_point_string_ entry (char(*), fixed bin, bit(*),
   fixed bin(35))   returns(float dec(59));
result = cv_fixed_point_string_ (fixed_point_string, default_base,
   switches, code);

Arguments:
result
   a float decimal(59) result produced by successfully converting
   fixed_point_string to a number. (Output)
fixed_point_string
   an ASCII representation of a fixed-point number ending with
   optional exponent and radix indicator sub-fields.  See the "Notes
   on input strings" and "List of radix indicators" sections.  If
   expressed as a decimal number, the string may include up to 59
   significant digits.  If expressed in another numeric base or radix,
   more digits may be given if base < 10; and fewer if base > 10.
   (Input)

default_base
   is the base in which the fixed_point_string is expressed if no
   radix indicator ends that string.  It must be a number between 2
   and 16 inclusive.  (Input)
switches
   control the action of this subroutine.  See the "Notes on switches"
   section.  (Input)

code
   standard status code indicating success of the convert operation.
   Failure values may be one of the following.
    error_table_$bad_arg
       default_base was not in the range:  2 <= default_base <= 16
    error_table_$bad_conversion
       an unexpected sign, digit, radix point or exponent was found in
       fixed_point_string; or more digits were given than will fit in
       the precision provided by a float dec(59) data type.
    error_table_$bigarg
       fixed_point_string exceeds implementation restriction of 256
       characters in length.
```

```
    error_table_$item_too_big
       an exponent overflow condition occurred when converting
       fixed_point_string to a float dec(59) value.
    error_table_$smallarg
       an exponent underflow condition occurred when converting
       fixed_point_string to a float dec(59) value.

Notes on switches:
The switches argument is assigned to a structure whose elements
ascribe meaning to the individual bits in switches.

   string(fixed_point_switches) = switches;

 dcl 1 fixed_point_switches based,
       2 signal_errors                 bit (1) unaligned,
       2 allow_exponent                bit (1) unaligned,
       2 convert_binary                bit (1) unaligned,
       2 convert_decimal               bit (1) unaligned;

This structure is declared in cv_fixed_point_string_.incl.pl1.

The structure elements are described in the "List of elements" section.
See the "List of named constants" section for named constants setting
bit values or combinations.

List of elements:
signal_errors
   "1"b: errors encountered during convert operation are signaled as
         PL/I conditions.  See the "Notes on condition handling"
         section.
   "0"b: errors are returned as status code values.

allow_exponent
   "1"b: fixed_point_string may include an exponent sub-field if
         expressed in a base in the range 2 through 13 inclusive.
         See the "Notes on input strings" section.
   "0"b: an exponent sub-field causes a conversion condition or status
         code.

convert_binary
   "1"b: use PL/I convert built-in to convert a fixed_point_string
         expressed as binary (base 2) digits.  See the "Notes on
         convert built-in" section.
   "0"b: use cv_fixed_point_string_ algorithms for a binary string.

convert_decimal
   "1"b: use PL/I convert built-in to convert a fixed_point_string
         expressed as decimal (base 10) digits.  See the "Notes on
         convert built-in" section.
   "0"b: use cv_fixed_point_string_ algorithms for a decimal string.

List of named constants:
   The following constants may be OR'd together or used individually
   as a value for the switches argument.
FIXED_POINT_SIGNALS
   convert operation errors are signaled as PL/I conditions.
FIXED_POINT_EXPONENT
   fixed_point_string may include an exponent sub-field.
FIXED_POINT_CONVERT_BIN
   binary strings are processed using the PL/I convert built-in.
```

```
FIXED_POINT_CONVERT_DEC
   decimal strings are processed using the PL/I convert built-in.

FIXED_POINT_EXPONENT_CONVERT_DEC
   turns on allow_exponent and convert_decimal.
FIXED_POINT_EXPONENT_CONVERT
   turns on allow_exponents, convert_binary and convert_decimal.

FIXED_POINT_SIG_EXP
   turns on signal_errors and allow_exponent.
FIXED_POINT_SIG_EXP_CONVERT_DEC
   turns on signal_errors, allow_exponent and convert_decimal.
FIXED_POINT_SIG_EXP_CONVERT
   turns on all four of the bits above.

List of radix indicators:
   A radix indicator sub-field may appear at the end of a
   fixed_point_string to indicate the base in which its digits are
   expressed.  Indicator characters may also be given in uppercase.
b, _b
   digits are interpreted as a base two number (binary).
q, _q
   digits are interpreted as a base four number (quaternary).
o, _o
   digits are interpreted as a base eight number (octal).
d, _d
   digits are interpreted as a base ten number (decimal).
x, _x
   digits are interpreted as a base sixteen number (hexadecimal).

rN, _rN
   digits are interpreted in the base N which is a decimal number
   where:  2 <= N <= 16

Notes on input strings:
A fixed_point_string argument consists of three sub-fields:

   +123456789.0034E+02_d
   \_ mantissa__ /\_/ \/
                  |  |
         exponent -'  '- radix indicator

The fixed_point_string may begin if an optional number of space (SP)
characters.

By default, the fixed_point_string is assumed to be positive.  A
leading plus (+) is allowed.  A negative value is specified using a
leading minus (-) character:
   -23

The fixed_point_string continues with a sequence of digits, and may
include one radix point (.) character separating integer digits
preceding the point from fractional digits following the point.  This
optionally signed sequence of digits with optional radix point is
sometimes called the "mantissa" or "significand":
   -23.004

Digits for bases 11 through 16 may be entered using letters for digit
values beyond 9, with letters accepted in either uppercase or
lowercase.  Digits for the most common numeric bases are shown in the
table below, ordered in ascending value.  Digits for a base N lower
```

than 16 use the first N valid digits listed for base 16.

BASE	VALID DIGITS IN THAT BASE
2	0 1
8	0 1 2 3 4 5 6 7
10	0 1 2 3 4 5 6 7 8 9
12	0 1 2 3 5 5 6 7 8 9 A B
16	0 1 2 3 4 5 6 7 8 9 A B C D E F

If fixed_point_switches.allow_exponent is "1"b, a fixed_point_string expressed in bases 2 through 13 begins with a mantissa which may be followed by an exponent sub-field. The exponent begins with one of the exponent indicator letters: D E F. The exponent digits are an optionally-signed decimal number N in the range:
-128 <= N <= 127

Exponents are not accepted for bases 14 through 16 because the exponent indicator letter is treated as a valid digit in those bases. No whitespace may appear between sign and digits of the exponent, or between mantissa and the exponent.

A positive exponent shifts the radix point right EXP places; a negative exponent shifts the radix point left EXP places. The following table shows decimal numbers with an exponent. Numbers expressed in another base would use that base to the power EXP as the multiplier.

STRING	mantissa * BASE**EXP	RESULT
123E-4	123 * 10**-4	0.0123
1.23E+1	1.23 * 10**+1	12.3
1.23E5_d	1.23 * 10**+5	123000
.123E+2d	.123 * 10**+2	12.3

A radix indicator sub-field may end the fixed_point_string to specify its numeric base. The default_base is assumed if no radix indicator is given. Use an indicator beginning with underscore (_) if the default_base includes the indicator letter as a valid digit; the underscore prevents a "b" or "d" radix indicator from being treated as a digit of the mantissa.

INDICATOR	NUMERIC BASE	EXAMPLES	VALUE (base 10)
b or _b	base 2 or binary	10001.11b	17.75
q or _q	base 4 or quaternary	113_q	23
o or _o	base 8 or octal	21.6o	17.75
d or _d	base 10 or decimal	17.75d	17.75
x or _x	base 16 or hexadecimal	11.Cx	17.75
rN or _rN	N is a decimal base number: 2 <= N <= 16	43r5	23
		15.9_r12	17.75

The fixed_point_string may end with one or more space (SP) characters. These characters are ignored by the conversion.

Notes on condition handling:
If fixed_point_switches.signal_errors = "1"b, the following PL/1 conditions are signaled if an error is encountered during the conversion.

CONDITION	EVENT TRIGGERING THE CONDITION
conversion	- whitespace characters other than space (" "). - space characters anywhere in the string other than the very beginning or end of string. - more than one sign character (+ or -) in the mantissa or exponent. - an invalid character in the mantissa. - more significant digits in the mantissa than will fit in a float dec(59) data type. - using an exponent sub-field when not allowed. - a non-decimal number in an exponent sub-field. - an invalid radix indicator string.

CONDITION	EVENT TRIGGERING THE CONDITION
overflow	- a string has significant digits in the mantissa plus an exponent sub-field causing the result exponent to be > +127.
underflow	- a string has significant digits in the mantissa plus an exponent sub-field causing the result exponent to be < -128.

Significant digits in the mantissa are those that follow any leading zero (0) digits. Leading zeroes are ignored. The table below shows how many significant digits can be stored in a float dec(59) data type for each numeric base. A mantissa with more significant digits than this number will trigger a conversion condition.

BASE	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MAX_DIGITS_ALLOWED	199	125	99	85	77	70	66	62	59	57	55	53	52	51	49

Just before the underflow or overflow condition is signaled, the current value of the oncode built-in function is pushed down and a fixed bin(35,0) code giving the reason for the signaling the condition is assigned to "oncode". The underflow and overflow conditions may not be restarted.

Just before the conversion condition is signaled, the current values of the oncode, onsource and onchar built-in functions are pushed down. A fixed bin(35,0) code giving the reason for the signaling conversion is assigned to "oncode". The string being converted is assigned to "onsource". The leftmost character for which the conversion failed is assigned to "onchar".

The caller's on-unit for any of these conditions may also call the cv_condition_$message subroutine to get additional information about that condition; or call cv_condition_$display to display such information to the user. For details, type: help cv_condition_

The conversion on-unit may perform a non-local goto to a label in the calling program to abort the failed convert operation. Or the convert operation may be restarted by having the on-unit use the onchar pseudo-variable to change the value of the leftmost failing character; or use the onsource pseudo-variable to change the entire contents of the string being converted. If the on-unit then ends without calling the continue_to_signal_ subroutine, the convert operation is retried using any replaced string values.

Notes on convert built-in:
If fixed_point_switches.convert_binary and/or .convert_decimal
switches are "1"b, cv_fixed_point_string_ uses the PL/I convert
built-in function to convert a binary and/or decimal string to a
float dec(59) data type.

The PL/I convert built-in function provides program access to PL/I
runtime operators that convert character strings to arithmetic data
types. These convert operators accept numeric strings expressed in
binary or decimal digits. Binary strings must end with a "b" radix
indicator. "b" is the only radix indicator accepted by the PL/I
convert operators. cv_fixed_point_string_ will add a "b" radix
indicator to an input string with base selected by default_base = 2,
or selected by another radix indicator form: _b or _r2 or r2

The convert operators are more efficient than cv_fixed_point_string_,
and accept binary and decimal strings in a wider variety of styles
than cv_fixed_point_string_. This includes support for complex number
strings including both real and imaginary parts (e.g., 123.4+32.0i).
The convert built-in may employ conversion techniques that generate
fewer exponent overflow or underflow conditions than the
cv_fixed_point_string_ algorithms.

Support for a wider variety of input styles comes with programming
costs: the program must be prepared to: explain accepted input styles
in its documentation; and handle additional conversion errors that can
occur when converting these added input styles. These issues are
avoided if the PL/I convert built-in bits in switches are off.

New Subroutine: cv_condition_.pl1

Using PL/I techniques for converting numeric strings to arithmetic data types provides more informative oncode messages and onchar location in the description of a conversion error. However, such details are difficult for a program's on-unit to obtain and display, and therefore are used only occasionally.

To make conversion and other arithmetic errors easier to report, a new cv_condition_ subroutine will be added. cv_condition_$display is a convenience routine that can be called directly from a conversion, size, fixedoverflow, overflow or underflow on-unit to display details of the failure in a 1- or 2-line message. cv_condition_$message provides detailed information as separate arguments which facilitate selective display of this information by an on-unit.

For conversion-related signals, the message includes an oncode number and description, the onsource string being converted, and the onchar digit or symbol being processed when then condition occurred. For example, when caller_name is "plus", the command might display the following message when trying to convert the string: 123.4*9

```
plus:  conversion condition:  oncode: 403  Invalid character follows a numeric field.
  onsource: "123.4*9"  onchar: "*"  onchar_index:  6
                ^
```

Notice the caret (^) pointing to the onchar character which was identified as significant in triggering the conversion error. It points to pl1_info.onchar_index, the onsource character index for the onchar built-in function value.

An overflow error occurs when trying to convert the string: 123E+200

```
plus:  overflow  condition:  oncode: 705  exponent overflow condition
```

Such conditions stem from arithmetic expressions that are usually trapped by the hardware. Neither onsource nor onchar values are available.

Documentation for the new subroutine entry points is shown in the cv_condition_.info segment below.

```
2021-12-26  cv_condition_

Function: a convenience utility for PL/I on-units, this subroutine
obtains and optionally displays information about conditions that may
occur when converting the ASCII representation of a number to an
arithmetic data type.

Entry points in cv_condition_:
 2021-12-26  cv_condition_$display
 2021-12-26  cv_condition_$message


Entry:  cv_condition_$display   (68 lines in entry point; 1 other entry point
in info)
```

2021-12-26 cv_condition_$display

Function: a convenience utility for PL/I on-units, this subroutine obtains and displays information about the most recently-signaled condition. Information is displayed on the user_output switch.

The subroutine calls find_condition_info_ to obtain information about the most recent signal. If a condition from the following list was signaled, then detailed information will be displayed.

```
   conversion       overflow    underflow
   fixedoverflow    size
```

For other condition names, the condition_interpreter_ subroutine is called to obtain a detailed message to display.

Syntax:

```
   declare cv_condition_$display entry (char(*), char(32) var);
   call cv_condition_$display (caller_name, condition_name);
```

Arguments:

caller_name
   string that identifies the caller. If not an empty string, the string followed by a colon begins the displayed message. (Input)

condition_name
   names a specific condition whose signaling information is to be displayed. Nothing is displayed for other conditions. Input an string to display information for any condition name. Upon return, it is set to the name of the condition that was found. If an empty string is returned, no message was found/displayed. (Input/Output)

Notes on displayed message:
This subroutine is designed to display detailed information about signals occurring when converting a numeric string to an arithmetic data type. For other conditions, it calls condition_interpreter_ to obtain a detailed description of the condition.

The following restartable conditions normally return to the point of error with no message displayed. This subroutine also displays nothing for these errors.

```
     command_error
     command_question
     finish
     stringsize
```

```
Examples:
For conversion-related signals, the message includes an oncode number
and description, the onsource string being converted, and the onchar
digit or symbol being processed when then condition occurred.  For
example, when caller_name is "plus", the command might display the
following message when trying to convert the string:  123.4*9

plus:  conversion condition:  oncode: 403  Invalid character follows a
numeric field.
  onsource: "123.4*9"  onchar: "*"  onchar_index:  6
                  ^
Notice the caret (^) pointing to the onchar character which was
identified as significant in triggering the conversion error.  It
points to pl1_info.onchar_index, the onsource character index for
the onchar built-in function value.


An overflow error occurs when trying to convert the
string:  123E+200

plus:  overflow  condition:  oncode: 705  exponent overflow condition

Such conditions stem from arithmetic expressions that are usually
trapped by the hardware.  Neither onsource nor onchar values are
available.


Entry:  cv_condition_$message   (110 lines in entry point; 1 other entry
point in info)

2021-12-26  cv_condition_$message

Function: a convenience utility for PL/I on-units, this subroutine
obtains information about the most recently-signaled condition.

The subroutine calls find_condition_info_ to obtain information about
the most recent signal.  If the condition was signaled while
converting an ASCII representation of a numeric string to an
arithmetic data type, details from the pl1_info structure attached to
that signal are formatted as a message.

For other conditions, the condition_interpreter_ subroutine is called
to obtain a message.

Syntax:
declare cv_condition_$message entry options(variable);
call cv_condition_$message (message, condition_name,
   oncode, oncode_message, onsource, onchar_index, onchar,
   condition_info_ptr);

Arguments:
   All arguments are positional.  All arguments except message are
   optional.  Include only arguments for the information needed.
   Use empty string or 0 placeholder values for intervening arguments
   whose values are not needed.
```

message
   a multi-line character string containing details of the most recently-signaled condition. Message text includes the oncode, oncode_message, onsource, onchar_index, and onchar values if such information is provided for the signaled condition. (Output)
condition_name
   name of the most recently-signaled condition. (Output)

oncode
   a numeric code value assigned by PL/I to the type of error that caused the condition to be signaled. For error types not diagnosed by PL/I runtime code or support procedures, this value is 0. This is the same value returned by the PL/I oncode built-in function. (Output)
oncode_message
   a string associated with the oncode value that describes the particular error causing a PL/I-related condition. For error types not diagnosed by PL/I runtime code or support procedures, this message is an empty string. (Output)

onsource
   for a conversion condition, this contains the character string being converted to an arithmetic data type when the condition occurred. For other conditions, this is an empty string. This is the same string returned by the PL/I onsource built-in function. (Output)

onchar_index
   for a conversion condition, this is the index within onsource of the character being processed when the condition occurred. For an oncode affecting the entire string being converted, it is the index of the final character in the string. For other conditions, it is 0. (Output)
onchar
   for a conversion condition, this is the character within the onsource which was being processed when the conversion condition occurred. For other conditions, it is an empty string. This is the same character returned by the PL/I onchar built-in function. (Output)

condition_info_ptr
   points to the condition_info structure returned by the find_condition_info_ subroutine for the most recently-signaled condition. This structure is declared in condition_info.incl.pl1. (Output)

Notes on message:

This subroutine returns detailed information about signals occurring when converting a numeric string to an arithmetic data type. For other conditions, it returns information provided by condition_interpreter_ for the condition.

```
The following restartable conditions normally return to the point of
error without displaying a message.  An empty string is returned for
these conditions:

     command_error
     command_question
     finish
     stringsize

Examples:
For conversion-related signals, the message includes an oncode number
and description, the onsource string being converted, and the onchar
digit or symbol being processed when then condition occurred.  For
example, for a conversion condition while trying to convert the
string:  123.4*9

conversion condition:  oncode: 403  Invalid character follows a numeric
field.
  onsource: "123.4*9"  onchar: "*"  onchar_index:  6
                  ^
Notice the caret (^) pointing to the onchar character which was
identified as significant in triggering the conversion error.  It
points to pl1_info.onchar_index, the onsource character index for
the onchar built-in function value.

An overflow error occurs when trying to convert the
string:  123E+200

overflow  condition:  oncode: 705  exponent overflow condition

Such conditions stem from arithmetic expressions that are usually
trapped by the hardware.  Neither onsource nor onchar values are
available.
```

Because cv_condition_ will be called by the plus family of command/AFs, this subroutine will be added to the bound_multics_bce_ bound segment containing plus.

Change: hex.pl1

The hex.pl1 program implements several command/active functions: binary (bin), octal (oct), decimal (dec), and hexadecimal (hex). This program currently accepts a fixed-point input string ending with optional single-letter radix indicator: b, q, o, x, or u (for unspec).

Unspec conversion accepts any input string entered by the user from 1 to 8 characters long, uses the unspec built-in function to capture the ASCII encoding for those characters, stores those bits in a fixed bin(71) variable, then converts that variable to the base selected by the command/AF name. For example, the string ABC is encoded in ASCII as shown below in the octal representation of those three characters.

```
octal ABCu
101102103
```

The current program does not accept the d radix indicator, nor the rN indicator, nor radix indicators beginning with an underscore.

Conversion code in this program will be replaced by a call to the new cv_fixed_point_string_ subroutine. That subroutine converts the input string to a float dec(59) value using radix_indicator_string_ to identify the radix indicator in all its forms; and it returns error_table_$bad_conversion if an error occurs during conversion. If hex.pl1 receives that error, it will check for the u radix indicator, and continue to support the unspec conversion in local code.

The simple changes shown in the compare_ascii output below will allow these command/AFs to support all standard radix indicator formats, and will simplify the code in hex.pl1.

```
cpa [lpn hex.pl1] ==

Inserted in B:
B11
B12       /****^  HISTORY COMMENTS:
B13         1) change(2021-12-05,GDixon):
B14            A) Remove code for converting integer string to float dec(59)
B15               (with support for bases 2, 4, 8, 10, 16 via radix indicator:
B16                b q o x).
B17            B) Replace with call to cv_fixed_point_string_ which does same
B18               type of conversion, and supports additional radix indicator
B19               formats.
B20            C) For binary and octal command/AFs with "u" (for unspec) input
B21               value, supply leading 0's for binary/octal digits zero-suppressed
B22               by numeric_to_ascii_base_.
B23                                                        END HISTORY COMMENTS */
B24
Preceding:
A11       hexadecimal: hex: proc;


Inserted in B:
B49       dcl  required_unspec_chars fixed bin;
B50       dcl  NO_UNSPEC_CHARS fixed bin int static options(constant) init(0);
Preceding:
A35       dcl (arg_len, return_len) fixed bin (21);
```

```
Inserted in B:
B65       dcl  cv_fixed_point_string_ entry (char(*), fixed bin, bit(1) aligned, bit(1) aligned,
fixed bin(35))
B66                 returns(float dec(59));
Preceding:
A49       dcl (ioa_, ioa_$nnl) entry options (variable);


A52       dcl (convert, decimal, index, low, substr, unspec) builtin;
Changed by B to:
B70       dcl  WHITESPACE  char (2) static options (constant) init ("           ");     /* SP HT
*/
B71
B72       dcl (copy, divide, index, low, substr, unspec) builtin;


A104                on conversion, overflow, underflow go to BAD_NUM;
Deleted by B, preceding:
B124
B125                do arg_index = 1 to arg_count;


Inserted in B:
B126                      required_unspec_chars = NO_UNSPEC_CHARS;     /* Setup of non-unspec
output.                             */
Preceding:
A107
A108                      call cu_$arg_ptr (arg_index, arg_ptr, arg_len, code);


A110                      if arg_len = 0 then float59 = 0;
A111                      else do;
A112
A113                           i = index ("bqox", substr (arg, arg_len, 1));
A114                           if i ^= 0 then do;
A115                                input_base = 2 ** i;
A116                                float59 = 0;
A117                                point_count = -1;                 /* scanning for decimal
point */
A118
A119                                do i = 1 to arg_len - 1;
A120
A121                                     if substr (arg, i, 1) = "." then do;
A122                                          if point_count >= 0 then do;
A123      BAD_NUM:                                 call complain (error_table_$bad_conversion, ME,
"^a", arg);
A124                                               return;
A125                                          end;
A126                                          point_count = 0;
A127                                     end;
A128                                     else do;
A129                                          digit_val = index ("0123456789ABCDEFabcdef", substr
(arg, i, 1)) - 1;
A130                                          if digit_val < 0 then go to BAD_NUM;
A131                                          if digit_val > 15 then digit_val = digit_val - 6; /*
lowercase abcdef */
A132                                          if digit_val >= input_base then go to BAD_NUM;
A133
A134                                          if point_count >= 0 then point_count = point_count +
1; /* after the decimal point */
A135
A136                                          float59 = float59 * decimal (input_base, 2) +
decimal (digit_val, 2);
A137                                     end;
A138                                end;
A139
A140                                if point_count > 0 then           /* decimal point in the
input */
A141                                     float59 = float59 / decimal (input_base ** point_count);
```

```
A142                         end;
A143                         else if substr (arg, arg_len, 1) = "u" then do; /* unspec */
A144                              arg_len = arg_len - 1;
A145                              if arg_len > 8 then do;
A146                                   call complain (0, ME, """u"" conversion only allows 8
characters.  ^au", arg);
A147                                   return;
A148                              end;
A149                              char8 = low (8 - arg_len) || arg;
A150                              unspec (fixed71) = unspec (char8);
A151                              float59 = fixed71;
A152                         end;
A153                         else float59 = convert (float59, arg);
Changed by B to:
B130      dcl  BASE_10 fixed bin int static options (constant) init (10);
B131      dcl  ENABLE_E_FORMAT bit (1) aligned int static options (constant) init ("1"b);
B132                                                              /* For base-10 inputs, E-
format number_string allowed.    */
B133                                                              /*  Conversion actually
done by PL/I convert builtin.     */
B134                                                              /*  E-format is not
meaningful in other numeric bases.     */
B135      dcl  ERRORS_NOT_SIGNALED bit (1) aligned int static options (constant) init ("0"b);
B136                                                              /* Return errors via code
argument.  These command/AFs      */
B137                                                              /*  are used in BCE
environment.  Some of the signal       */
B138                                                              /*  apparatus is not
present in that environment.           */
B139
B140                      float59 = cv_fixed_point_string_ (arg, BASE_10, ERRORS_NOT_SIGNALED,
ENABLE_E_FORMAT, code);
B141                                                              /* Let subroutine do
conversion work.  It handles */
B142                                                              /*  radix chars more
completely while still using PL/I     */
B143                                                              /*  convert built-in to do
the actual conversion of    */
B144                                                              /*  decimal input; or doing
its own conversion for     */
B145                                                              /*  non-decimal input
bases.                                 */
B146
B147                      if code = error_table_$bad_conversion &
B148                         substr (arg, arg_len, 1) = "u" then do;   /* cv_fixed_point_string_
does not support "u" radix      */
B149                           arg_len = arg_len - 1;                 /*  indicator.  Continue to
support that conversion here. */
B150                           if arg_len > 8 then do;
B151                                call complain (0, ME, """u"" conversion only allows 8
characters.  ^au", arg);
B152                                return;
B153                           end;
B154                           char8 = low (8 - arg_len) || arg;
B155                           unspec (fixed71) = unspec (char8);
B156                           float59 = fixed71;
B157
B158                           if  base = 2  then                     /* Binary output requires 9
bits for each unspec(char)    */
B159                                required_unspec_chars = arg_len * BITS_PER_CHAR;
B160                           else if  base = 8  then                /* Octal output requires 3
digits for each unspec(char)   */
B161                                required_unspec_chars = arg_len * divide( BITS_PER_CHAR, 3,
17, 0 );
B162                      end;
B163                      else if code ^= 0 then do;
B164                           if (index(WHITESPACE, substr(arg, 1, 1)) > 0) |
B165                              (index(WHITESPACE, substr(arg, arg_len, 1)) > 0) then
B166                                call complain (code, ME, """^a""", arg);
```

```
B167                          else call complain (code, ME, "^a", arg);
B168                          return;

Inserted in B:
B172                       if  required_unspec_chars > 0  then          /* Supply leading 0's for
bin/oct unspec output.           */
B173                          val_str = copy("0", required_unspec_chars - length(val_str))  ||
val_str;
Preceding:
A157      APPEND:


A166
Changed by B to:
B183      %page;
B184      %include system_constants;
B185

Comparison finished: 9 differences, 113 lines.
```

Change: plus.pl1

The plus.pl1 program implements several command/active functions: plus, minus, times, divide, mod, max, min, quotient, ceil, trunc, floor. This program currently accepts decimal fixed-point numbers, and uses the PL/I convert built-in function to convert its arguments to the float dec(59) data type.

The program will be changed to call the new cv_fixed_point_string_ to convert arguments to the float dec(59) data type. This will permit inputs that end with a radix indicator sub-field.

The program will call cv_condition_$display to report to display conversion conditions to the user.

The change also adds a round command/AF entry point to plus.pl1. This implements the PL/I round built-in function at command level, thus providing an active function to shorten fractions with 59-character precision to more human-friendly lengths.

The changes are shown in the compare_ascii output below.

```
cpa [lpn plus.pl1] ==

Inserted in B:
B13       /****^  HISTORY COMMENTS:
B14         1) change(1974-01-27,BWolman),approve(),audit(),install():
B15            Initial version by Barry L. Wolman.
B16         2) change(1976-07-08,Herbst),approve(),audit(),install():
B17            Fixed min and others to operate with no args.
B18         3) change(2021-12-05,GDixon):
B19             A) Use cv_fixed_point_string_ to convert input numbers to float dec(59)
B20                data type, thereby supporting inputs in non-decimal bases.
B21             B) Add the round command/active function.
B22                                                        END HISTORY COMMENTS */
B23
Preceding:
A13       /* Arithmetic active functions


A27
A28          Each Ai is the character string representation of a valid PL/I decimal number,
A29          either fixed or float.  Calculations are performed internally using float dec(59)
A30          arithmetic.  The result is in I-, F-, or E-format depending on its value.  All of
A31          these active functions can be called as functions or as commands, in which case
A32          they print the result.
A33
A34          Initial Version: 27 January 1974 by Barry L. Wolman */
A35
A36       /* Modified 7/8/76 by S. Herbst */
A37       /* Fixed min and others with no args 07/07/81 S. Herbst */
Changed by B to:
B38                round A1 A2                  round(A1, A2)
B39
B40          Each Ai is the character string representation of either:
B41            - a valid PL/I decimal number (either fixed or float), or
B42            - a valid fixed-point number expressed in a base from 2 to 16.
B43
B44          Calculations are performed internally using float dec(59) arithmetic.
B45
B46          The result is in I-, F-, or E-format depending on its value.
B47
B48          All of these active functions can be called as functions or as commands,
B49          in which case they print the result.
B50
B51          ------------------------------------------------------------------------ */
```

```
Inserted in B:
B57                 prec_fixed fixed bin,
Preceding:
A43                 result char(72) varying,


A52                 (mod,max,min,fixed,convert,string,trunc,floor,ceil) builtin,
Changed by B to:
B67                 (mod,max,min,fixed,string,trunc,floor,ceil,round) builtin,


Inserted in B:
B70       dcl       BASE_10 fixed bin int static options (constant) init (10);
B71
Preceding:
A55       dcl       (cu_$arg_ptr,cu_$af_arg_ptr,cu_$af_return_arg) entry(fixed bin,ptr,fixed
bin,fixed bin(35)),


A57                 numeric_to_ascii_ entry(float dec(59),fixed bin,char(72) varying),
A58                 (ioa_,com_err_,active_fnc_err_) options(variable);
Changed by B to:
B74                 cv_fixed_point_string_ entry (char(*), fixed bin, bit(*), fixed bin(35))
returns(float dec(59)),
B75                 numeric_to_ascii_ entry(float dec(59),fixed bin,char(72) varying),
B76                 (ioa_,com_err_,active_fnc_err_) options(variable),
B77                 gripe entry variable options(variable);


A64                 2 multi   bit(1) unaligned,
A65                 2 unary   bit(1) unaligned;
A66       \014
Changed by B to:
B83                 2 multi   bit(1) unaligned,                     /* More than 2 arguments
supported.                       */
B84                 2 unary   bit(1) unaligned;                     /* Only 1 argument allowed.
*/
B85       %page;


A130      \014
Changed by B to:
B149
B150      round:    entry;
B151
B152                op = "round";
B153                string(op_type) = "00"b;
B154                goto join;
B155      %page;


Inserted in B:
B164                     gripe = com_err_;
Preceding:
A139                     code = 0;


Inserted in B:
B171                     gripe = active_fnc_err_;
Preceding:
A145                     end;


A159      simple_err:    if not_active_function then call com_err_ (code, op);
A160                     else call active_fnc_err_ (code, op);
Changed by B to:
B186      simple_err:    call gripe (code, op);
```

```
A164                on conversion goto not_numeric;
Changed by B to:
B190      /*        on conversion goto not_numeric;*/


A169                call get_arg(1,ap,al,code);
A170
A171                if code ^= 0 then call gripe("");
A172
A173                number1 = convert(number1,arg);
Changed by B to:
B195        dcl  cv_condition_$display entry (char(*), char(32) var);
B196                on conversion begin;
B197                     call cv_condition_$display( (op), "" );
B198                     goto exit;
B199                     end;
B200
B201                call get_arg(1,ap,al,code);
B202                if code ^= 0 then do;
B203                     call gripe(code, op, "Numeric argument.");
B204                     goto exit;
B205                     end;
B206
B207                number1 = cv_fixed_point_string_( arg, BASE_10,
FIXED_POINT_SIG_EXP_CONVERT_DEC, code );


A186
A187                     if code ^= 0 then call gripe("");
A188
A189                     number2 = convert(number2,arg);
Changed by B to:
B220                     if code ^= 0 then do;
B221                          call gripe(code, op, "Second numeric argument.");
B222                          goto exit;
B223                          end;
B224
B225                     number2 = cv_fixed_point_string_( arg, BASE_10,
FIXED_POINT_SIG_EXP_CONVERT_DEC, code );


Inserted in B:
B235                     if op = "round" then do;                    /* PL/1 round built-in
accepts only a constant 2nd op     */
B236                                                                 /*  between 1 and 59 when
1st operand is float dec(59).   */
B237                          number2 = abs(trunc(number2));         /*  - Convert number2 to
non-negative integer.           */
B238                          number2 = min(59.0, number2);          /*     which is between 0
and 59                          */
B239                          prec_fixed = number2;                  /*  - Convert to fixed bin
data type required by          */
B240                          goto ROUND(prec_fixed);                /*     goto
LABEL(array_index)                          */
B241
B242      ROUND(1):           number1 = round(number1,  1); goto output;
B243      ROUND(2):           number1 = round(number1,  2); goto output;
B244      ROUND(3):           number1 = round(number1,  3); goto output;
B245      ROUND(4):           number1 = round(number1,  4); goto output;
B246      ROUND(5):           number1 = round(number1,  5); goto output;
B247      ROUND(6):           number1 = round(number1,  6); goto output;
B248      ROUND(7):           number1 = round(number1,  7); goto output;
B249      ROUND(8):           number1 = round(number1,  8); goto output;
B250      ROUND(9):           number1 = round(number1,  9); goto output;
B251      ROUND(10):          number1 = round(number1, 10); goto output;
B252      ROUND(11):          number1 = round(number1, 11); goto output;
B253      ROUND(12):          number1 = round(number1, 12); goto output;
B254      ROUND(13):          number1 = round(number1, 13); goto output;
B255      ROUND(14):          number1 = round(number1, 14); goto output;
```

```
B256      ROUND(15):          number1 = round(number1, 15); goto output;
B257      ROUND(16):          number1 = round(number1, 16); goto output;
B258      ROUND(17):          number1 = round(number1, 17); goto output;
B259      ROUND(18):          number1 = round(number1, 18); goto output;
B260      ROUND(19):          number1 = round(number1, 19); goto output;
B261      ROUND(20):          number1 = round(number1, 20); goto output;
B262      ROUND(21):          number1 = round(number1, 21); goto output;
B263      ROUND(22):          number1 = round(number1, 22); goto output;
B264      ROUND(23):          number1 = round(number1, 23); goto output;
B265      ROUND(24):          number1 = round(number1, 24); goto output;
B266      ROUND(25):          number1 = round(number1, 25); goto output;
B267      ROUND(26):          number1 = round(number1, 26); goto output;
B268      ROUND(27):          number1 = round(number1, 27); goto output;
B269      ROUND(28):          number1 = round(number1, 28); goto output;
B270      ROUND(29):          number1 = round(number1, 29); goto output;
B271      ROUND(30):          number1 = round(number1, 30); goto output;
B272      ROUND(31):          number1 = round(number1, 31); goto output;
B273      ROUND(32):          number1 = round(number1, 32); goto output;
B274      ROUND(33):          number1 = round(number1, 33); goto output;
B275      ROUND(34):          number1 = round(number1, 34); goto output;
B276      ROUND(35):          number1 = round(number1, 35); goto output;
B277      ROUND(36):          number1 = round(number1, 36); goto output;
B278      ROUND(37):          number1 = round(number1, 37); goto output;
B279      ROUND(38):          number1 = round(number1, 38); goto output;
B280      ROUND(39):          number1 = round(number1, 39); goto output;
B281      ROUND(40):          number1 = round(number1, 40); goto output;
B282      ROUND(41):          number1 = round(number1, 41); goto output;
B283      ROUND(42):          number1 = round(number1, 42); goto output;
B284      ROUND(43):          number1 = round(number1, 43); goto output;
B285      ROUND(44):          number1 = round(number1, 44); goto output;
B286      ROUND(45):          number1 = round(number1, 45); goto output;
B287      ROUND(46):          number1 = round(number1, 46); goto output;
B288      ROUND(47):          number1 = round(number1, 47); goto output;
B289      ROUND(48):          number1 = round(number1, 48); goto output;
B290      ROUND(49):          number1 = round(number1, 49); goto output;
B291      ROUND(50):          number1 = round(number1, 50); goto output;
B292      ROUND(51):          number1 = round(number1, 51); goto output;
B293      ROUND(52):          number1 = round(number1, 52); goto output;
B294      ROUND(53):          number1 = round(number1, 53); goto output;
B295      ROUND(54):          number1 = round(number1, 54); goto output;
B296      ROUND(55):          number1 = round(number1, 55); goto output;
B297      ROUND(56):          number1 = round(number1, 56); goto output;
B298      ROUND(57):          number1 = round(number1, 57); goto output;
B299      ROUND(58):          number1 = round(number1, 58); goto output;
B300      ROUND(59):          number1 = round(number1, 59); goto output;
B301      ROUND(0):           end;
Preceding:
A199
A200                     end;


A214                if not_active_function then call com_err_(0,op,"Attempt to divide by zero.");
A215                else call active_fnc_err_(0,op,"Attempt to divide by zero.");
A216                return;
A217
A218      not_numeric:
A219                call gripe("""^a"" is non-numeric");
A220                return;
A221
A222      too_big:
A223                call gripe("overflow");
A224                return;
A225
A226      too_small:
A227                call gripe("underflow");
A228                return;
A229
A230      gripe:          proc(s);
A231
```

```
A232      dcl            s char(*);
A233
A234                     if not_active_function then call com_err_(code,op,s,arg);
A235                     else call active_fnc_err_(code,op,s,arg);
A236
A237                     goto exit;
A238                     end;
A239
A240      exit:     end;
Changed by B to:
B317                call gripe(0,op,"Attempt to divide by zero.");
B318                goto exit;
B319
B320      not_numeric:
B321                call gripe(0,op,"""^a"" is non-numeric", arg);
B322                goto exit;
B323
B324      too_big:
B325                call gripe(0,op,"Exponent overflow");
B326                goto exit;
B327
B328      too_small:
B329                call gripe(0,op,"Exponent underflow");
B330                goto exit;
B331
B332      exit:     return;
B333      %page;
B334      %include cv_fixed_point_string_;
B335
B336      end plus;
B337

Comparison finished: 15 differences, 221 lines.
```

Change: calc.pl1

The calc.pl1 source implements a command/AF that performs several calculator-like operations according to an arithmetic expression given as an argument. When invoked as a command without arguments, the calc subsystem reads statements from the user, evaluating each statement and displaying the result. Statements may be assignment of an expression to a user variable, or just an expression to be evaluated.

calc.pl1 was written before the EIS instruction set was added to Multics hardware. It uses float bin(27) data type for its calculations.

This change upgrades calc to use the float dec(59) data type for all its calculations, and to call cv_fixed_point_string_ to convert numeric strings in the expressions to float dec(59) data values; and use numeric_to_ascii_ to convert numbers for display. This will permit support of numeric strings including a radix indicator sub-field, and will permit more precise calculations over a wider data exponent range.

calc will call cv_condition_$message from its on-units to report conversion, size, overflow and underflow conditions to the user.

Changes to calc.pl1 are shown in as compare_ascii output below.

```
cpa [lpn calc.pl1] ==

A89                3 value float bin (27);
A90
A91       dcl  ffip entry (ptr, fixed bin (17), fixed bin (17), float bin (27));
A92       dcl  ffop entry (char (32) aligned, fixed bin (17), float bin (27));
Changed by B to:
B109               3 value float dec (59);
B110
B111      dcl  cv_condition_$message entry() options(variable);
B112      dcl  cv_fixed_point_string_ entry (char(*), fixed bin, bit(*), fixed bin(35))
returns(float dec(59));
B113      dcl  numeric_to_ascii_ entry (float dec (59), fixed bin) returns (char (72) var);


A95       dcl  iox_$get_line entry (ptr, ptr, fixed bin, fixed bin, fixed bin (35));
A96       dcl  iox_$put_chars entry (ptr, ptr, fixed bin, fixed bin (35));
A97       dcl  iox_$user_output ptr ext;
A98       dcl  iox_$user_input ptr ext;
A99       dcl  cu_$cp entry (ptr, fixed bin, fixed bin (35));
A100      dcl  cu_$grow_stack_frame entry (fixed bin (17), ptr, fixed bin (35));
A101      dcl  (noprt, ileq) bit (1);
A102      dcl  funcs (0:6) char (8) static internal init ("sin", "cos", "tan", "atan", "abs",
"ln", "log");
A103      dcl  (abs, addr, atan, cos, fixed, index, length, log, log10, ltrim) builtin;
A104      dcl  (mod, null, rtrim, sin, substr, tan, verify) builtin;
A105
A106      dcl  (fixedoverflow, overflow, program_interrupt, underflow) condition;
A107                                                                 /*\014               */
A108
Changed by B to:
B116      dcl  iox_$get_line entry (ptr, ptr, fixed bin (21), fixed bin (21), fixed bin (35));
B117      dcl  iox_$user_input ptr ext static;
B118      dcl  cu_$cp entry (ptr, fixed bin (21), fixed bin (35));
B119      dcl  cu_$grow_stack_frame entry (fixed bin, ptr, fixed bin (35));
B120      dcl  (noprt, ileq) bit (1);
B121
B122      dcl  funcs (0:6) char (8) int static options (constant) init ("sin", "cos", "tan",
"atan", "abs", "ln", "log");
B123      dcl  (abs, addcharno, addr, after, atan, before, cos, fixed, index, length, log, log10,
ltrim,
B124           maxlength, mod, null, rtrim, search, sin, size, substr, tan, verify) builtin;
```

```
B125
B126      dcl  (conversion, fixedoverflow, overflow, program_interrupt, underflow) condition;
B127      %page;


A132            vars.d.value (0) = 3.14159265e0;
A133            vars.d.name (1) = "e";
A134            vars.d.value (1) = 2.7182818e0;
Changed by B to:
B151            vars.d.value (0) =
3.14159265358979323846264338327950288419716939937510582097494e0;
B152                                                                  /* From www.math.com  (59
digits of precision) */
B153            vars.d.name (1) = "e";
B154            vars.d.value (1) =
2.71828182845904523536028747135266249775724709369995957496696e0;
B155                                                                  /* From math.utah.edu (59
digits of precision) */


A140            on overflow, fixedoverflow begin;
A141               error_string = "Overflow";
A142               go to HANDLE_FAULT;
A143            end;
A144            on underflow begin;
A145               error_string = "Exponent too small";
A146               go to HANDLE_FAULT;
A147            end;
A148
A149      new_line: ss = -1;                                          /* reinitialize variables
*/
Changed by B to:
B161            on conversion, overflow, size, underflow begin;
B162               call cv_condition_$message( error_string, condition_name );
B163
B164               if af_sw then call active_fnc_err_ (0, "calc", "^a", error_string);
B165               else call ioa_$ioa_switch (iox_$error_output, "^a", error_string);
B166               if expr_arg_sw then go to RETURN_FROM_AF;
B167               else go to new_line;
B168            end;
B169
B170      new_line:
B171            ss = -1;                                              /* reinitialize variables
*/


A153               call cu_$grow_stack_frame (104, vp, code);         /* if vars too big, get
more space */
Changed by B to:
B175               call cu_$grow_stack_frame (size (vars), vp, code); /* if vars too big, get
more space */


A165
A166               begin;
A167      dcl  expr_arg char (arg_len + 1);
A168
A169                  expr_arg = arg || "
A170      ";
A171                  call prec_calc (expr_arg, arg_len + 1, dum, code);
A172
A173               end;
A174
Changed by B to:
B187               call prec_calc (arg || NL, arg_len + 1, dum, code);
B188      RETURN_FROM_AF:
```

```
A182               call ioa_ ("CALC 1.1");
Changed by B to:
B196               call ioa_ ("CALC 2.0");

A195
A196      HANDLE_FAULT:
A197            if af_sw then call active_fnc_err_ (0, "calc", "^a", error_string);
A198            else call ioa_$ioa_switch (iox_$error_output, "^a", error_string);
A199            if expr_arg_sw then return;
A200            else go to new_line;
A201                                                              /*\014                  */
Changed by B to:
B209      %page;


A210      dcl  (i, j, k, num, last, level, ip, strt) fixed bin (17);
A211      dcl  code fixed bin (35);
A212      dcl  (x, fval) float bin (27);
Changed by B to:
B218      dcl  (i, j, k, num, last, level, ip, strt) fixed bin (21);
B219      dcl  code fixed bin (35);
B220      dcl  (x, fval) float dec (59);


A297                  s.value (ss - 3) = s.value (ss - 3) ** fixed (s.value (ss - 1), 17, 0);
Changed by B to:
B305                  s.value (ss - 3) = s.value (ss - 3) ** fixed (s.value (ss - 1), 17, 0);


A319               call ffop (out, ip, fval);                         /* convert value to char
string */
A320               return_string = rtrim (ltrim (substr (out, 1, ip - 1)));
Changed by B to:
B327               out = numeric_to_ascii_ (fval, 0);
B328               return_string = rtrim (ltrim (out));


A326            substr (out, 1, 5) = "=   ";                          /* set up output line */
A327            call ffop (out, ip, fval);                            /* convert value to char
string */
A328            substr (out, ip, 1) = "
A329      ";                                                          /* append NL to output line
*/
A330            call iox_$put_chars (iox_$user_output, addr (out), ip, (0));
Changed by B to:
B334            out = numeric_to_ascii_ (fval, 0);                    /* set up output line */
B335            call ioa_ ("=   ^a", out);
B336


A358               call ffip (addr (in), num - 1, ip, s.value (ss));  /* if numeric then call
ffip for conversion */
Changed by B to:
B364               call get_number (addr (in), num - 1, ip, s.value (ss));


A464               wrk = "
A465      ";                                                          /* set wrk = NL */
A466               call iox_$put_chars (iox_$user_output, addr (wrk), 1, (0)); /* print a NL */
Changed by B to:
B470               call ioa_ ("");
```

```
A470                  substr (out, 1, 8) = vars.d.name (j);
A471                  substr (out, 9, 4) = " =  ";
A472                  ip = 13;
A473                  call ffop (out, ip, vars.d.value (j));          /* call ffop to convert
value to char string */
A474                  substr (out, ip, 1) = "
A475      ";                                                          /* insert NL */
A476                  call iox_$put_chars (iox_$user_output, addr (out), ip, (0));
Changed by B to:
B474                  out = numeric_to_ascii_ (vars.d.value (j), 0);
B475                  call ioa_ ("^va =  ^a", maxlength (vars.d.name (j)), vars.d.name (j), out);


A547
Changed by B to:
B546      /* format: off */
B547      %page;
B548      /* ------------------------------------------------------------------------------
B549         QUICK INTERNAL PROCEDURE:  get_number
B550
B551         Function:  Locates end of the next number_string in input line (var: in) and
B552                    passes number_string to cv_fixed_point_string_ to convert
B553                    it to a float dec(59) data scalar.
B554
B555         Formats supported:  Valid numbers accepted, expressed as Perl patterns (regular
expressions).
B556                             Note that leading sign is processed by prec_calc routine as a
unary operator.
B557                             So leading sign (while permitted) is not shown in the following
patterns.
B558
B559                    Simplest:  [0-9A-Fa-f.]+                      ( Just a fixed-point
numeric string, no exponent )
B560
B561         w/ Radix Indicator:  [0-9A-Fa-f.]+_?[BQODXbqodx]         ( fixed-point string
followed by radix indicator )
B562                            |   [0-9A-Fa-f.]+_?[Rr][0-9]+
B563
B564                 w/ Exponent:  [0-9.]+[Ee][+-]?[0-9]+             ( E-format exponent allowed
IFF base-10 )
B565                            |   [0-9.]+[Ee][+-]?[0-9]+_?[Dd]
B566                            |   [0-9.]+[Ee][+-]?[0-9]+_?[Rr]10
B567
B568
B569         The following characters break the sequences of characters that can appear in a
B570         number string:
B571
B572               Ending Break:  [) =+-/*]                          ( skip over pattern:
[Ee][+-]
B573                                                                   which occurs in E-format
exponent
B574                                                                   of a possible decimal
number string )
B575
B576         ------------------------------------------------------------------------------ */
B577      /* format: on */
B578
B579      get_number:                                                 /* Convert next input token
in input line to a number     */
B580         proc (inP, inL, inX, number);
B581
B582      dcl  in char (inL) based (inP),
B583           inP ptr,
B584           inL fixed bin (21);
B585      dcl  inX fixed bin (21);
B586      dcl  number float dec (59);
B587
B588      dcl  BREAK_CHARS char (7) int static options (constant) init (") =+-/*");
B589      dcl  breakL fixed bin (21) init (0);
```

```
B590
B591      dcl  number_string char (number_stringL) based (number_stringP),
B592           number_stringP ptr,                                    /* Candidate to be the
numeric token                      */
B593           number_stringL fixed bin (21);
B594      dcl  numberX fixed bin (21);                                /* Index into number_string
of next BREAK_CHAR           */
B595            number_stringP = addcharno (inP, inX - 1);
B596            number_stringL = inL - (inX - 1);
B597
B598      dcl  exponent_string char (exponent_stringL) based (exponent_stringP),
B599           exponent_stringP ptr,                                  /* Candidate for exponent
in an E-format number string.   */
B600           exponent_stringL fixed bin (21);
B601      dcl  exponentX fixed bin (21);
B602
B603            numberX = search (number_string, BREAK_CHARS);        /* Find possible end of the
numeric_string token.         */
B604            if numberX > 0 then do;
B605               if (substr (number_string, numberX, length ("+")) = "+" |
B606                    substr (number_string, numberX, length ("-")) = "-") then do;
B607                                                                  /*  - If BREAK_CHAR is + or
-, it could be sign in the    */
B608                                                                  /*    exponent part of E-
format numeric string.          */
B609
B610                  if (substr (number_string, numberX - 1, length ("E")) = "E" |
B611                       substr (number_string, numberX - 1, length ("e")) = "e") then do;
B612                     exponent_stringP = addcharno (number_stringP, numberX);
B613                     exponent_stringL = length (number_string) - numberX;
B614                                                                  /*    Overlay just the
exponent part of the number.       */
B615                     number_stringL = numberX;                    /*    number_string
overlays mantissa of E-format number. */
B616
B617                     exponentX = search (exponent_string, BREAK_CHARS);
B618                     if exponentX > 0 then                        /*  - Next BREAK_CHAR
definitely ends number_string token.*/
B619                          exponent_stringL = exponentX - 1;
B620                     number_stringL = length (number_string) + length (exponent_string);
B621                                                                  /*    Recombine mantissa
and exponent parts.              */
B622                  end;
B623
B624                  else number_stringL = numberX - 1;              /*  - Not in E-format so
BREAK_CHAR ends number_string    */
B625               end;
B626               else number_stringL = numberX - 1;                 /*  - Not + or -, so
BREAK_CHAR ends number_string.       */
B627            end;
B628
B629      dcl  BASE_10 fixed bin int static options (constant) init (10);
B630      dcl  SIGNAL_ERRORS bit (1) aligned int static options (constant) init ("1"b);
B631                                                                  /* Use onsource/onchar to
better diagnose location of     */
B632                                                                  /*  conversion error in
number_string.                    */
B633      dcl  ENABLE_E_FORMAT bit (1) aligned int static options (constant) init ("1"b);
B634                                                                  /* For base-10 inputs, E-
format number_string allowed.    */
B635                                                                  /*  Conversion actually
done by PL/I convert builtin.     */
B636                                                                  /*  E-format is not
meaningful in other numeric bases.    */
B637            inX = inX + length (number_string);
B638            number = cv_fixed_point_string_ (number_string, BASE_10,
FIXED_POINT_SIG_EXP_CONVERT_DEC, code);
B639
B640         end get_number;
```

```
B641      %page;
B642      %include cv_fixed_point_string_;
```

The calc program and its two support routines ffip and ffop are the only components of the bound_calc_ bound segment. Both ffip and ffop are being eliminated by this change. This change proposes moving calc into bound_full_cp_ (where hex/dec/oct/bin command/AFs reside). The changes for this addition are shown in the compare_ascii output below.

```
cpa [lpn bound_full_cp_.bind -lb o] ==

Inserted in B:
B87         6) change(2021-12-05,GDixon):
B88            Move calc into this bound segment.
Preceding:
A87                                                        END HISTORY COMMENTS */


Inserted in B:
B113                        calc,
Preceding:
A111                        rank,


Inserted in B:
B123                        calc,
Preceding:
A120                        decimal, dec,


Inserted in B:
B151      objectname:       calc;
B152       retain:          calc;
B153
Preceding:
A147      objectname:       decode_entryname_;

Comparison finished: 4 differences, 7 lines.
```

Change: interpret_info_struc_.pl1

The current message provided by interpret_info_struc_ includes both onsource and onchar strings associated with the condition, and a message showing the oncode value describing nature of the conversion error. Add a line below the onsource description pointing to the character position within that string that caused the error. This character position is obtained from the pl1_info.oncharindex value passed by PL/I when signaling conversion. With this change, a conversion condition description might be:

```
Error:  conversion condition by >udd>m>gd>w>calc>calc$prec_calc|3731 (line 625)
onsource = “+10H55.4E-2_d”, onchar = "H"
                ^
Invalid character follows a numeric field.
```

There is currently no PL/I built-in function which provides this location. Numeric strings will contain up to three components (signed mantissa, optional exponent, optional radix indicator) making it harder for users and developers to understand which component caused a particular conversion error. This added explanation helps overcome the difficulty.

PL/I conversion operators already provide the oncharindex value. cv_integer_string_, cv_fixed_point_string_, and radix_indicator_string_ will also provide that information when signaling a conversion condition.

Testing the New Software

Testing varies by type of source program and nature of the change to that program.

Testing radix_indicator_string_.pl1

radix_indicator_string_.pl1 is a new subroutine, so its interface and operation require thorough testing.

Test goals include the following.

A. Ensure that the radix indicator sub-field is properly distinguished from earlier parts of its numeric string.

B. Ensure that radix indicator strings that could be accepted as final digit of a non-decimal number are treated as part of that number and rejected as a radix indicator sub-field.

C. Ensure that all documented radix indicator string types work correctly.

D. Ensure that other radix indicator sub-field characters are diagnosed as errors.

This program is called by cv_fixed_point_string_ so testing will be integrated into the test exec_com for cv_fixed_point_string_.

Test results show the new code is operating correctly and meets all of its design objectives and the above testing goals.

Testing cv_condition_.pl1

cv_condition_.pl1 is new code providing useful messages describing the errors that may occur when converting numeric string inputs to arithmetic data types. It focuses on conversion conditions that report the majority of failures encountered in string-to-arithmetic conversion attempts. But it reports special data for other number-related conditions (overflow, underflow, size, fixedoverflow) and calls condition_interpreter_ to get system default messages for other kinds of conditions.

Test goals include the following.

A. Ensure messages for conversion conditions include all of the following data:
    - oncode number detailing the type of conversion failure.
    - oncode interpretation describing the type of conversion failure as an English message.
    - onchar built-in data showing the character being evaluated when the failure event was diagnosed.
    - onchar_index data pointing to the location of the onchar data within the onsource, so the user can understand which sub-field of the numeric input was being evaluated, and which character position within that field caused the failure when the onchar letter/digit appears multiple times in the input string.
    - onsource built-in data showing the actual string being converted with a pointer to the onchar letter/digit within that string.
B. Ensure messages for number-related conditions include an oncode number and interpretation if such information is provided with the condition.
C. Ensure that condition_interpreter_ is correctly invoked to provide messages for other conditions.

cv_condition_$display is called by plus.pl1. cv_condition_$message is called by calc.pl1. So calc and entry points in plus can be used to test messages returned by cv_condition_.

cv_condition_$display is called from the several programs described below for cv_fixed_point_string_ testing, and most of its testing happens as these higher-level programs test conversion of input strings and report failure results of such conversion attempts. In fact, cv_condition_ was designed to facilitate failure reporting by such test programs. It proved so easy and beneficial to use that the program should be installed for wider use.

Testing cv_fixed_point_string_.pl1

cv_fixed_point_string_.pl1 is new code trying to emulate the conversion strategy used by the any_to_any_ subroutine for converting numeric strings to real data types, with digits expressed in numeric bases from 2 through 16. Switches control the conversion as follows.

- **signal_errors**: report errors via signaled conditions vs. Multics static code.
- **allow_exponent**: accept fixed-point strings with an exponent sub-field vs. excluding exponent sub-field.
- **convert_decimal**: after evaluating any radix indicator sub-field to determine base of the input string, use the PL/I convert built-in function to evaluate the string with any radix indicator removed iff:
    - an exponent sub-field is accepted;
    - the string is expressed as decimal digits.
- **convert_binary**: after evaluating any radix indicator sub-field to determine base of the input string, use the PL/I convert built-in function to evaluate the string with any radix indicator replaced by a 'b' indicator required for PL/I binary inputs iff:
    - an exponent sub-field is accepted;
    - the string is expressed as binary digits.

Several kinds of testing were performed.

- *exploratory testing* determines the exact order in which any_to_any_ detects errors when several errors appear in an input string, and the oncode and onchar values returned as each type of error is detected.
- *range testing* determines the precision of mantissa inputs accepted and the range of exponents accepted in the returned float dec(59) result.
- *functional testing* checks of the results returned for typical input strings to ensure accuracy of the conversion for decimal inputs, and range of inaccuracy expected when converting non-decimal fractions to the decimal digits in a float dec(59) mantissa.

fld59.pl1: performs exploratory tests by calling the PL/I convert built-in to change that input string to a float dec(59) data type. The input string is displayed, following by results of successful convert, using numeric_to_ascii_ change the float dec(59) back to a display string. Conditions occurring during the conversion are trapped by on-units and displayed via cv_condition_$display.

Syntax as a command: fld59 FIXED_POINT_STRING

tcfps.pl1: performs range and functional tests by calling cv_fixed_point_string_ and reporting results using numeric_to_ascii_to report the raw float dec(59) result, and numeric_to_ascii_base_ to reconvert it back to its input base for a comparison display of accuracy.

```
Syntax as a command: tcfps DEFAULT_BASE SWITCHES ON-UNIT_CASE NUMBER_TO_CONVERT

 SWITCHES
  1      Report errors using signaled conditions.
  01     Allow number string with exponent sub-field.
  011    Use PL/I convert built-in for binary number string.
  0101   Use PL/I convert built-in for decimal number string.
  1111   With signals, exponent, convert.

 ON-UNIT_CASE
    0    Status code errors.
    1    Details from cv_condition_$display.
    2    Details from cv_condition_$display then cu_$cl.
    3    Details from cv_condition_$display then return to cv_fixed_point_string_.
    4    Details from default_error_handler_/condition_interpreter.
   1x    Like case 0-4 but don't call PL/I convert after testing
         cv_fixed_point_string_.
```

Its arguments allow choice of default numeric base, setting of switches to cv_fixed_point_string_, and setup of various error handling environments: status codes, signalled conditions, and different on-units to handle conditions expected in particular tests. Some of these on-units call cv_condition_$display to report errors; others allow the signal to be reported by default_error_handler_ to evaluate changes to the interpret_info_struc_.pl1 program supporting default_error_handler_ and condition_interpreter_. For comparison testing, the program has an option to use the PL/I convert built-in to perform a similar conversion of the exact input string, and report any results of the conversion.

tcfps.ec: a test suite containing test scripts to perform several different series of tests. Each section of the exec_com outlines an area of testing and runs a series of tests using one of the above testing programs. Comments displayed before/after individual tests tell what result is expected (if not obvious from the inputs to the test program).

```
ec t>tcfps TOC
Syntax as a command: ec tcfps {TEST_GROUP}

Arguments:
TEST_GROUP is one of the following labels:
    signs                       test error processing in mantissa/exponent
                                  leading sign
    exponent_range, range, er   test exponent range limit testing
    fractions, frac             test handling of fractional mantissa digits
    oncodes, onc                test all oncodes
     onc401                      no exponent digits
     onc402                      no number digits
     onc403                      invalid chars in number
      onc403_collate              test all chars in collate except controls,
                                    and 0 and 1 digits
     onc404                      many radix points in mantissa
     onc405                      long input string
     onc407                      many exponents in number
     onc408                      test exponent >  127
     onc409                      test exponent < -128
     onc413                      test more than 10 exponent digits
     onc414                      no mantissa digits
```

```
     onc418                        more than 59 significant mantissa digits
    fixed_point_order, order      test order in which cv_fixed_point_string_ finds
                                    errors
    radix_indicator, radix        test radix indicator sub-field for proper
                                    operation and errors
Exploratory Tests ------------
    PL/I_error_order, pl1_order   test order in which PL/I convert finds errors
    max_power_for_base, power     test largest  base ** POWER  value.
```

For example:

```
ec t>tcfps onc407
--- oncode = 407: many exponents --------------------


                            (More than one exponent sub-field in number.)
tcfps 10 (1100 1111) 11  1.23E+1D+2

tcfps ------- 10   in: "1.23E+1D+2"
 Sig Exp:  conversion condition:  oncode: 407  Too many exponential sub-fields in numeric field.
  onsource: "1.23E+1D+2"  onchar: "D"  onchar_index:  8
                    ^

tcfpsConvert- 10   in: "1.23E+1D+2"
 Sig Exp:  conversion condition:  oncode: 407  Too many exponential sub-fields in numeric field.
  onsource: "1.23E+1D+2"  onchar: "D"  onchar_index:  8
                    ^

tcfps 10 (1100 1111) 11  1.01E+1F+2b

tcfps ------- 10   in: "1.01E+1F+2b"
 Sig Exp:  conversion condition:  oncode: 407  Too many exponential sub-fields in numeric field.
  onsource: "1.01E+1F+2b"  onchar: "F"  onchar_index:  8
                    ^

tcfpsConvert- 10   in: "1.01E+1F+2b"
 Sig Exp:  conversion condition:  oncode: 407  Too many exponential sub-fields in numeric field.
  onsource: "1.01E+1F+2b"  onchar: "F"  onchar_index:  8
                    ^
```

Documentation Changes

Documentation for the new radix_indicator_string_ and cv_fixed_point_string_ subroutines was provided above. The following subsections show replacements for the functions provided by plus.pl1, hex.pl1 and calc.pl1.

plus.info

Documentation in plus.info describes all of the commands implemented in plus.pl1. This includes: plus, minus, times, divide, mod, max, min, quotient, ceil, trunc, floor, round. Change are focused on describing the radix indicator sub-field that can now end NUMBER arguments. An info block documents the new round command/AF

```
                    plus.info 12/05/21  2004.6 pst Sun


:Info: plus:  2021-12-05  plus

Syntax as a command:  plus {NUMBERs}


Syntax as an active function:  [plus {NUMBERs}]


Function:  returns the sum of NUMBERs.  If no NUMBERs are specified,
0 (the additive identity) is returned.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:     -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: minus:  2021-12-05  minus

Syntax as a command:  minus {NUMBERa {NUMBERb}}


Syntax as an active function:  [minus {NUMBERa {NUMBERb}}]


Function:  returns the result of NUMBERa minus NUMBERb.  If NUMBERb is
not specified, the negative of NUMBERb is returned.  If no arguments
are specified, returns 0.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:    -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: times:  2021-12-05  times

Syntax as a command:  times {NUMBERs}


Syntax as an active function:  [times {NUMBERs}]


Function:  returns the product of the NUMBERs.  If no NUMBERs are
specified, 1 (the multiplicative identity) is returned.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:   -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: divide:  2021-12-05  divide

Syntax as a command:  divide NUMBERa NUMBERb


Syntax as an active function:  [divide NUMBERa NUMBERb]


Function:  returns the integer part of the decimal quotient of NUMBERa
divided by NUMBERb.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:    -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.


Notes:  mod and quotient are related active functions.  To understand
this relationship type:  help mod quotient
```

```
:Info: quotient:  2021-12-05  quotient

Syntax as a command:  quotient NUMBERa NUMBERb


Syntax as an active function:  [quotient NUMBERa NUMBERb]


Function:  returns the result of NUMBERa divided by NUMBERb.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:    -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.


Notes:  mod and divide are related active functions.  To understand
this relationship type:  help mod divide
```

```
:Info: mod:  2021-12-05  mod

Syntax as a command:  mod NUMBERa NUMBERb


Syntax as an active function:  [mod NUMBERa NUMBERb]


Function:  returns the remainder of NUMBERa divided by NUMBERb (or
NUMBERa modulo NUMBERb).


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:    -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.


Notes:  divide and quotient are related active functions.  To
understand this relationship type:  help divide quotient
```

```
:Info: max:  2021-12-05  max

Syntax as a command:  max NUMBERs


Syntax as an active function:  [max NUMBERs]


Function:  returns the maximum of the numeric arguments passed to it.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:    -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: min:  2021-12-05  min

Syntax as a command:  min NUMBERs


Syntax as an active function:  [min NUMBERs]


Function:  returns the minimum of the numeric arguments passed to it.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:    -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: ceil:  2021-12-05  ceil

Syntax as a command:  ceil NUMBER


Syntax as an active function:  [ceil NUMBER]


Function:  returns the smallest decimal integer greater than or equal
to NUMBER.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:     -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: trunc:  2021-12-05  trunc

Syntax as a command:  trunc NUMBER


Syntax as an active function:  [trunc NUMBER]


Function:  returns the largest decimal integer whose absolute value is
less than or equal to the absolute value of NUMBER.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:     -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: floor:  2021-12-05  floor

Syntax as a command:  floor NUMBER


Syntax as an active function:  [floor NUMBER]


Function:  returns the largest decimal integer less than or equal to
NUMBER.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:     -128 <= EXPONENT <= +127


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.
```

```
:Info: round:  2021-12-05  round

Syntax as a command:  round NUMBER PRECISION


Syntax as an active function:  [round NUMBER PRECISION]


Function:  rounds NUMBER to PRECISION decimal places.


Arguments:
NUMBER
   a fixed-point number with precision up to 59 decimal places.
   NUMBER may end with an optional radix indicator to specify
   a value expressed in a non-decimal base.  See "List of radix
   indicators" below.  An E-Format exponent may be given with a
   decimal number in the range:     -128 <= EXPONENT <= +127
PRECISION
   a decimal integer between 1 and 59 indicating the number of
   significant decimal places to retain when rounding NUMBER at the
   PRECISION decimal place.


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive


:hcom:
/****^  HISTORY COMMENTS:
  1) change(2021-12-05,GDixon):
      A) Combine infos for the following related functions into a single
         multi-block info segment:  plus  minus  times  divide  mod
                                    max  min  quotient  ceil  trunc
                                    floor  round
      B) Add documentation for the round command/active function.
                                                  END HISTORY COMMENTS */
```

hex.info

Documentation for the commands in hex.pl1 is shown below in binary.info. This includes blocks documenting the command/AFs: binary (bin), octal (oct), decimal (dec) and hexadecimal (hex).

```
:Info: binary: bin:  1983-03-31  binary, bin

Syntax as a command:  bin VALUEs


Syntax as an active function:  [bin VALUEs]


Function:  returns one or more numeric or character strings in binary
(numeric base 2).


Arguments:
VALUE
   is an input string to be processed.  It is usually a fixed-point
   decimal string (e.g., 21 or 55.9).  It may be a numeric string
   ending with a radix indicator which specifies a non-decimal base.
   See "List of radix indicators" below.  It may be an arbitrary
   character string (1 to 8 characters long) followed by a "u"
   (for "unspec") indicator.  See "Notes on unspec values" below.


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.


Notes on unspec values:
A VALUE argument ending with a "u" indicator is treated as character
data.  The PL/I unspec built-in converts each ASCII character to its
binary encoding for the character.  Those binary bits are then
displayed.  VALUE can be from 1 to 8 characters long.  This permits
the user to see the binary encoding for the given characters.


Examples:
binary 21
10101
```

```
binary 55.9
110111.111001100110011001100110011001100110011001100110011

binary 30o
11000

binary AB_x
10101011


binary ABCu
001000001001000010001000011

binary (A B C)u
001000001
001000010
001000011
```

```
:Info: octal: oct:  1983-03-31  octal, oct

Syntax as a command:  oct VALUEs


Syntax as an active function:  [oct VALUEs]


Function:  returns one or more numeric or character strings in octal
(numeric base 8).


Arguments:
VALUE
   is an input string to be processed.  It is usually a fixed-point
   decimal string (e.g., 21 or 55.9).  It may be a numeric string
   ending with a radix indicator which specifies a non-decimal base.
   See "List of radix indicators" below.  It may be an arbitrary
   character string (1 to 8 characters long) followed by a "u"
   (for "unspec") indicator.  See "Notes on unspec values" below.


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.

Notes on unspec values:
A VALUE argument ending with a "u" indicator is treated as character
data.  The PL/I unspec built-in converts each ASCII character to its
binary encoding for the character.  Those binary bits are then
displayed.  VALUE can be from 1 to 8 characters long.  This permits
the user to see the binary encoding for the given characters.


Examples:
octal 21
15

oct 55.9
67.7146314631463146314631463146314631463146314631463146314631463

oct 30d
36

oct AB_x
```

```
253


oct 1.E2
144

oct ABCu
101102103
```

```
:Info: decimal: dec:  1983-03-31  decimal, dec

Syntax as a command:  dec VALUEs


Syntax as an active function:  [dec VALUEs]


Function:  returns one or more numeric or character strings in decimal
(numeric base 10).


Arguments:
VALUE
   is an input string to be processed.  It may be a fixed-point
   decimal string (e.g., 21 or 55.9), but is often a numeric string
   ending with a radix indicator which specifies a non-decimal base.
   See "List of radix indicators" below.


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.


Examples:
decimal 21_d
21

dec 55.7o
45.875

dec 33q
15

decimal AB_x
171
```

```
:Info: hexadecimal: hex:  1983-03-31  hexadecimal, hex

Syntax as a command:  hex VALUEs

Syntax as an active function:  [hex VALUEs]

Function:  returns one or more numeric or character strings in
hexadecimal (numeric base 16).

Arguments:
VALUE
   is an input string to be processed.  It may be a fixed-point
   decimal string (e.g., 21 or 55.9), but is often a numeric string
   ending with a radix indicator which specifies a non-decimal base.
   See "List of radix indicators" below.

List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).

rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive.

Examples:
hexadecimal 21
15

hex 55.7o
2d.e

hex 33q
f

hex AB_x
a

:hcom:
/****^  HISTORY COMMENTS:
  1) change(2021-12-05,GDixon):
      A) Combine binary.info octal.info decimal.info and hexadecimal.info
         into a single multi-block info segment.  All these commands share
         same interface and should be documented together.
      B) Remove documentation for "u" input format indicator from
         decimal.info and hexadecimal.info.  The unspec(char_string)
         functionality doesn't have much meaning in dec/hex output base.
                                                  END HISTORY COMMENTS */
```

calc.info

The following documentation for the calc command/AF has been updated to reflect the change to the float dec(59) data type.

```
                    calc.info 12/05/21  2107.8 pst Sun


:Info: calc:  2021-12-07  calc

Syntax as a command:  calc {EXPRESSION}


Syntax as an active function:  [calc EXPRESSION]


Function: provides a calculator capable of evaluating arithmetic
expressions with operator precedence, a set of often-used functions,
and a memory that is symbolically addressable (i.e., by identifier).

Invoke the command without arguments to enter the calculator
subsystem.  The user may enter EXPRESSION lines, assign EXPRESSIONs to
VARIABLEs, list VARIABLEs, etc.  Enter a quit (q) request to exit the
subsystem.


Arguments:
EXPRESSION
   an arithmetic expression to be evaluated.  If this argument is
   specified, the calc command prints its value and returns to
   command level.  The expression must be quoted if it contains
   spaces or other command language characters.  Variables are not
   allowed.


Notes on expressions:
Arithmetic expressions involve real values and the operands +, -, *,
/, and ** (addition, subtraction, multiplication, division, and
exponentiation).  Unary (or prefix) plus (+) or minus (-) operators
are allowed.  Parentheses can be used, and blanks between operators
and values are ignored.  Calc evaluates each expression according to
rules of precedence and prints out the result.  The order of
evaluation is as follows:

  expressions within parentheses
  function references
  prefix +, prefix -
  **
  *, /
  +, -


For example, if you type

   2 + 3 * 4

calc responds

   = 14

Operations of the same level are processed from left to right except
```

```
for the prefix plus and minus, which are processed from right to left.
This means 2**3**4 is evaluated as (2**3)**4.
Notes on numbers:
Numbers may be entered as integers (123), or fixed point strings
(1.23).  Decimal numbers may include an E-format exponent
(1.23e+2, 1.23e2, 1.23E2, or 1230E-1).  Numbers may end with a radix
indicator (477o) to enter non-decimal values in bases 2 through 16.
See "List of radix indicators" below.

Numbers are stored as float dec(59) values.  A precision of 59 decimal
digits is maintained.  Exponent range: -128 <= EXPONENT <= +127


List of requests:
   In the calculator subsystem, the following requests may be used.
<EXPRESSION>
   prints out the value of expression.
<VARIABLE>=<EXPRESSION>
   assigns value of expression to variable.
list
   prints out the names and values of all the variables that have
   been declared so far.
<VARIABLE>
   prints out the name and value of variable.
q
   returns to command level.


.
   causes calc to identify itself by printing "calc".
..<COMMAND_LINE>
   causes the remainder of the line to be passed to Multics as a
   command line and executed.


Notes on assignment statements:
In the calculator subsystem, the value of an expression can be
assigned to a variable.  The form of an assisgnment is:

     <VARIABLE> = <EXPRESSION>

The VARIABLE name must begin with a letter, have a total length
of 8 or fewer characters, and must be made up of uppercase and/or
lowercase letters, digits, and the underscore character (_).


For example, the following are legal assignment statements--

     x = 35

     Rho = sin(2*theta)

The calc command does not print any response to assignment statements.


Notes on variables:
variables can be used in place of literal numbers.  For example:

    r = 32
    pi * r ** 2
```

```
Preset variables include:
    pi:   ratio of a circle's circumference to its diameter; and
     e:   Euler's number
The first 59 digits of these irrational values have been stored.


Notes on functions:
Seven functions are provided--sin, cos, tan, atan, abs, ln, and log
(ln is base e, log is base 10).  They can be nested to any level, e.g.,
sin(ln(var).5*pi/180).


List of radix indicators:
   Indicator characters may also be given in uppercase.
b, _b
   the number is interpreted as a base two number (binary).
   See "Notes" below.
q, _q
   the number is interpreted as a base four number (quaternary).
o, _o
   the number is interpreted as a base eight number (octal).
d, _d
   the number is interpreted as a base ten number (decimal).
   See "Notes" below.
x, _x
   the number is interpreted as a base sixteen number (hexadecimal).


rN, _rN
   the number is interpreted in the base N which is a decimal number
   between 2 and 16 inclusive
```