TO: MSPM Distribution
FROM: C. Garman
DATE: January 11, 1969
SUBJECT: Revisions to cv_string

Attached is a major revision of the documentation of cv_string. Note especially that an EPL call to

```
cv_string
```

is now equivalent to a call to

```
cv_string$cs.
```

Also of interest is a short tutorial on usage for scanning character strings.

Published: 01/11/69
(Supersedes: BY.10.03 06/23/67)

Identification

Virtual strings; in-line string manipulation and decomposition
cv_string
C. Garman

Purpose

The purpose of cv_string is (1) to provide the means for "converting" all or part of a (possibly varying) string into a fixed_length, "virtual string", without moving the string, (2) to create a virtual string given an arbitrary pointer plus offset and length information, (3) given a string, to provide pointer, length and offset information for in-line manipulation of short bit and character strings, and (4) given a pointer to an ACC-type string (ASCII Character with Count, BD.7.01, p3), to create a virtual string pointing to the character information only.

For (1), (2), and (4) above, "virtual-string" means that a string is "created", from thin air as it were, having all the properties of a string, but which in fact is an "overlay" on the data region of the source variable. The advantages of this approach are generally in the area of reduction of actual movement of strings, as well as in the performance of tasks which the EPL compiler does not support but which the EPL run-time routines do support. Needless to say, there are disadvantages, the prime one being mis-use, for which read the section titled Warning. Bon appétit!

The basic implementation makes use of a little-known feature of the EPL compiler, which is used in the implementation of the substr function/pseudo-variable, and as such is implementation dependent.

Usage:

1. "converting" strings

```
     dcl string {bit}(*),     /* (*) if parameter of procedure
                {char}           or (N) if user's own variable */

 virtual_string {bit}(*),     /* not a parameter */
                {char}

       (i,j) fixed bin(17);
```

To make virtual_string into the equivalent of string:

```
call cv_string$(bs|cs) (string, virtual_string);
```

To make virtual_string begin with the i-th bit/character of string:

```
call cv_string$(bs|cs) (string, virtual_string, i);
```

To make virtual_string contain j bits/characters beginning with the i-th bit/character:

```
call cv_string$(bs|cs) (string, virtual_string, i, j);
```

In the last two descriptions of usage, the value of i should be 1 to get the first bit/character of string into virtual_string; in case it is not obvious, the last description corresponds closely to the operation of the substr function and is also roughly equivalent to a dynamic occurence of

```
dcl virtual_string (bit|char) (j) defined string position (i);
```

in PL/I (C28-6571-4, pp 56-58), but note that in the PL/I definition i would have to be a decimal integer constant. In the cases above a call to cv_string is equivalent to a call to cv_string$cs.

2. Changing pointer to virtual string.

```
dcl stringp ptr;          /* other declarations as before */

    call cv_string$(bp|cp) (stringp, virtual_string, i, j);
```

This call is similar to the 4-argument call listed under (1) above, except that the first argument is a pointer directly to the data of a string, instead of a string variable (with its attendant specifier), and the first bit/character is assumed to be in the first bit/character position of the word pointed to by stringp. (Note numbering of bits is 1-36 in a word, not 0-35; similarly characters are numbered 1-4, not 0-3.)

3. Extracting pointer information from string.

call cv_string$${bx \atop cx}$$ (string, stringp, i, j)

For these calls, string is, as before, an arbitrary bit or character string, either varying or non-varying, and may be a parameter of the procedure or an arbitrary string variable. The arguments stringp, i, and j are all output arguments and must be supplied.

Upon return, stringp points to the first word containing any of the data, i is the offset in bits or characters of the first relevant bit/character in the word pointed to by stringp (the range is 0-35 for bit strings, 0-3 for characters), (the range is different from (1) and (2) due to considerations of accessing efficiency, and for historical reasons dealing with the numeration of elements of a string), and j is the number of bits/characters in string.

4. Get virtual string from pointer to ACC-type character-string.

```
dcl accptr ptr;          /* points to first word of
                            ACC-string */

    call cv_string$accp(accptr, virtual_string);
```

(An ACC-type string is a string of characters whose first 9 bits (1st character) are interpreted as the count of the succeeding characters.)

Upon return virtual_string represents the data pointed to by accptr.

Implementation

The segment is coded in EPLBSA.

On entry various switches are set: the size in bits of the basic element (1 for $bs, $bp, $bx, 9 for $cv_string, $cs, $cp, $cx, and $accp) and whether the result is a pointer ($bx, $cx,) or a new specifier (all others).

Where the input argument is a string, the specifier is examined to check for legality, and the various indirections performed to get the real address of the data, as well as the bit offset and (current) length.

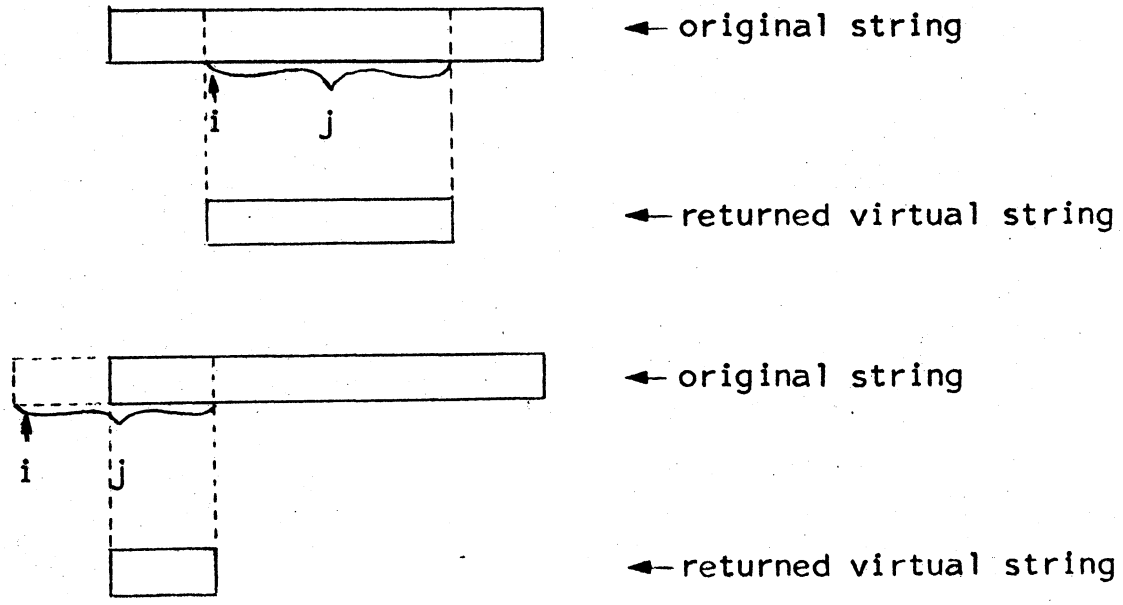
At this point, if pointer information is to be returned, the program renormalizes the length and offset information to the appropriate element size, returns the pointer, offset and length variables, and returns to its caller.

If only two arguments were passed to the $cv_string, $cs or $bs entries, control passes to the cleanup portion of the program, which computes the bit offset and data pointer values, adds the ID bits, stores away the revised dope and the new data pointer, and returns. (Note again that no data movement has taken place, and that virtual_string "points to" the same data as the original input string or pointer [or to a sub_string of it]).

If four arguments were passed to the three entries named above or if control comes in at the $cp or $bp entries, the value of j and the original length of the input string is used as the basis for further calculation (if the input was a pointer, a value of 36 x 2**18 bits is assumed for the original length of the string - the longest non-overlapping bit string containable in one segment).

If j is negative or zero, further computation ceases and a zero-length string results.

Next the value of i is picked up and tests are performed for cases of the requested string overlapping the bounds of the input string; in no case will the returned virtual string overlap the bounds of the original input string (see examples in Fig. 1).



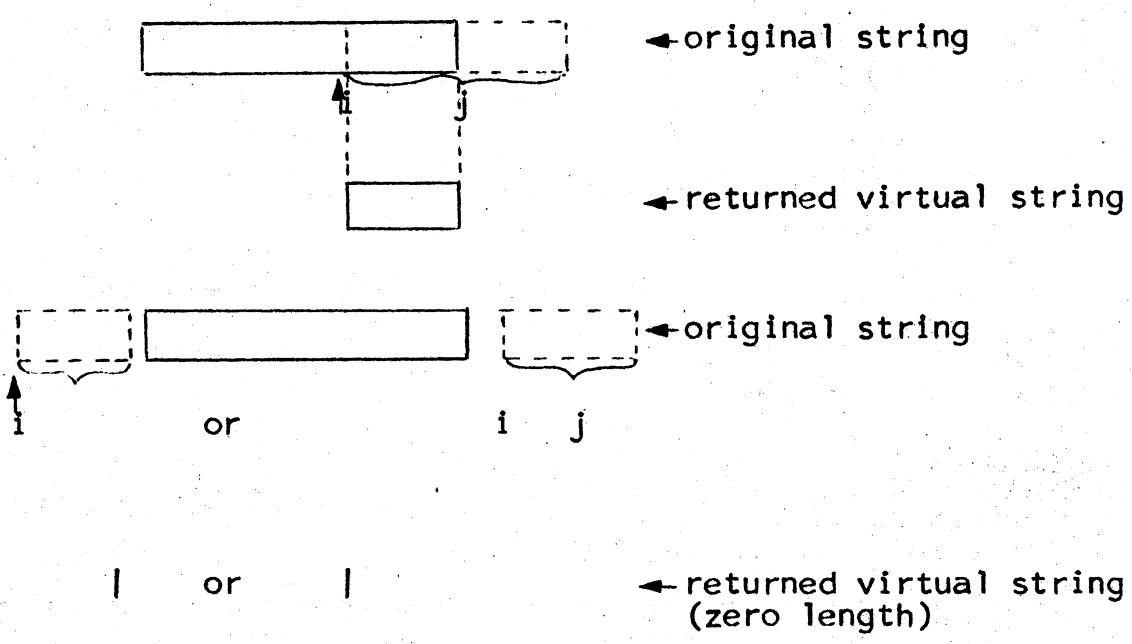


Figure 1

Finally, control passes to the cleanup section for return of the specifier for whatever residual virtual string remains, and control returns to the caller.

For the $accp entry, the first 9 bits of the word pointed to in the first argument are picked up, the value is multiplied by 9 (the number of bits per character) and control passes immediately to the cleanup code.

Notes

Except for the $accp entry, the data words of the input string or pointer are never examined.

No checks are made as to whether the effective string would wrap-around the end of a segment (i.e., whether the word-offsets 777777(8) and 0 are contiguous in the resultant virtual string).

Likewise, no check is made of the legality of the first character of the ACC-string, since the maximum length that would result is a 511-character string.

Error Checking

The table below lists error conditions which may occur:

1. invalid code in specifier of first argument.

2. $cs, $cx only: number of bits in first argument not evenly divisible by 9, or bit-offset not divisible by 9.

When an error in the arguments is encountered a zero-op-code is executed; the address field contains the given error code, which shows up in the EA word of the SCU data for the resultant fault.

Warning

Certain rules should be followed to avoid erroneous, if not fatal, mis-use of cv_string.

For instance, a varying string can be passed in a calling sequence as a non-varying one, by calling cv_string, after, and only after, the last assignment statement dealing with that string between the "definition" and the use of virtual_string in a call, it will most likely point to trash as its data, since varying strings keep the data in a free-storage area, and a new assignment changes the location of the physical allocation of the string.

In a similar vein, if a procedure wishes to return a virtual string (passed by a parameter) to its caller, care must be taken that the virtual string does not point to data that will be part of an inactive block when control returns, that is, the data must not be part of an automatic variable in the called procedure's stack frame. (This is equivalent to saving a pointer to an automatic variable in a descendent block) (Note that a string literal may be the subject of a cv_string: its lifetime is effectively "eternal", since it is in the text segment; i.e., it should remain active throughout the subsequent life of the process). Likewise, for the entries to cv_string where the first argument is a string, it should be a real variable, and not an expression (e.g., the concatenation of two variables), since EPL is likely to re-use the temporary string created to hold the resultant string with deleterious results as mentioned for varying strings.

When a virtual string is the object of an assignment statement (i.e., on the "left-hand side" of the "=" sign), no information remains as to whether the data was or was not originally part of a varying string; thus the PL/I rules for assignment to a non-varying string apply, with padding or truncation as appropriate.

Examples

The following program is an example of some uses of cv_string for scanning a line of the form

arg1 arg2 arg3 ... argn

After an initialization call to "prime" the procedure, calls to the main entry-point extract arguments (for simplicity in the example, only 1 space is allowed between arguments, and none at the beginning).

When the end of the line is reached, a zero length string is returned.

```
scan_line$prime: proc(line);          /* initialization
                                         entry */

dcl  line char(*),                    /* may be varying */

     (scanp ptr,
      (si, sl, so) fixed bin(17)) int static;
                                      /* save values */

     call cv_string$cx(line, scanp, so, sl);

     si=0;

     return;

fetch_arg: entry(arg);                /* return virtual-
                                         string representing
                                         atomic argument */

dcl  arg char(*);                     /* a virtual string
                                         will be returned */

dcl  1 tx based (scanp),
       2 ch (0:1023) char(1),

     do i = si to (sl-1) while (scanp->tx.ch(i+so) ¬= " ");
     end;

     call cv_string$cp(scanp, arg, si+so+1, i-si);
     si=i + 1;

end scan_line$prime;
```

The following program purports to call scan_line correctly:

```
test_scan: proc;
dcl  val char(*);
     call scan_line$prime("a bc def ghij klmno");
get_val: call scan_line$fetch_arg(val);
     if length (val) ^= 0;
     then do;
          call write_out$nl("val =`"||val||"'.", 0);
          go to get_val;
          end;
          call write_out$nl("All done", 0);
end test_scan;
```

Note the following features:

1. The declaration used to access the characters is a 2 level structure, since EPL will not directly pack single-level arrays of strings. The array information is placed with the character string, however, on the expectation of such a millenium.

2. The array bounds are set starting at 0 to remove the need for 6 instructions of testing code per reference to the array, having to do with virtual origins, with the resultant need to add 1 in the call to cv_string$cp. (The 6 instructions could be replaced by 1 instruction placed a little more strategically, but that is a different matter.)

3. The built-in length function may be used to examine the returned virtual string; this is preferable to

   ```
   if val = " "
   ```

   since the latter results in an external call to accomplish the comparison (the compiler does not know at compile-time how long the string will be; it does know, however, how to compile code to find out what the length is, if the program wishes that information).

4. The string may be used in any legal PL/I expressions in a normal way; note again, however, that the concatenation shown will result in creation of two temporary varying strings.

Where calls are made to modules which require non-varying strings as arguments (such as to the basic file system), the following coding sequences maybe used if one has on hand only varying strings:

```
dcl (v1, v2 . . .) char(N) varying;
                                        /* N is generally numeric.
                                           e.g., 511 */

dcl (x1, x2 . . .) char(*);             /* not varying */

     call cv_string(v1, x1);

     . . .

     call cv_string (vn, xn);

     call hcs_$fgxxby (x1, x2 . . .);
```

While less esthetic, the following also works:

```
dcl (v1, v2, ...)char(N)var;            /* as before */

dcl cv_string ext entry
    returns (char(*));

     call hcs_$fgxxby(cv_string (v1), cv_string(v2), . . .);
```

Further usages, grotesque or otherwise, are up to the fertile imagination of the user; perusal of code successfully using cv_string is highly recommended.