To: Distribution

From: Paul .Green

Date: 12/27/79

Subject: Recent Improvements to PL/I

INTRODUCTION

The Multics PL/I compiler has recently been modified to improve the performance of some old constructs, and several new constructs have been added that are more efficient than the old ones they replace. The purpose of this memo is to publicize these recent additions, so that programmers can convert old programs, and can use the constructs in new programs.

NEW BUILTINS

clock and vclock Builtins

The clock and vclock builtin functions have been added to the language to provide a fast and easy way to get the real-time clock, and the virtual-time clock, respectively. The following table gives the old and new methods:

```
OLD:    declare clock_ entry () returns (fixed bin (71));
        c = clock_ ();

NEW:    declare clock builtin;
        c = clock ();

OLD:    declare virtual_cpu_time_ entry ()
                  returns (fixed bin (71));
        vc = virtual_cpu_time_ ();

NEW:    declare vclock builtin;
        vc = vclock ();
```

Since both the old and new functions return a fixed bin(71) value, no compatibility problems can arise if the old external function was properly declared. If either clock_ or virtual_cpu_time_ was declared to return "fixed bin (52)" or "bit (72)", for example, then extreme care must be taken when

converting to the new builtins. I have already found a bug in IOI that was caused by the following incorrect conversion:

```
OLD:      declare clock_ entry () returns (fixed bin (52));
          bit32 = bit (clock_ (), 32);

NEW, WITH BUG:
          declare clock builtin;
          bit32 = bit (clock (), 32);

NEW, WITH BUG FIXED:
          declare clock builtin;
          bit32 = substr (bit (clock (), 71), 20, 32);
  OR:
          bit32 = substr (bit (bin (clock (), 52), 52), 1, 32);
  OR:
          bit32 = bit (bin (clock (), 52), 32);
```

A call to clock_ takes 51 microseconds; a call to clock takes 16 microseconds. A call to virtual_cpu_time_ takes 37 microseconds; a call to vclock takes 22 microseconds. The object segment is slightly smaller, since no link needs to be generated. The dynamic linking overhead is also reduced; from one snapping per program, to one snapping per process.

The clock and vclock builtins are available in Release 24 and subsequent releases.

rank Builtin

The rank builtin function converts a char(1) value to a fixed bin(9) value that represents the binary encoding of the character. In other words, it converts a character to its ASCII value (it also works on non-ASCII characters, those > 177 octal).

The rank builtin function has been added to the language to replace two equivalent constructs, one of which was efficient, but machine-dependent, the other of which was inefficient, but machine-independent. The rank builtin function is both efficient and machine-independent. The following old constructs should be changed to use the rank builtin:

OLD	NEW
binary (unspec (char1), 9)	rank (char1)
index (collate9 (), char1)-1	rank (char1)
binary (unspec (substr (cs, i, 1)), 9)	rank (substr (cs, i, 1))

In the 3rd case given above, the new implementation of rank generates substantially better code than the old way.

The rank builtin is available in Release 25 and subsequent releases.

byte Builtin

The byte builtin converts a fixed bin (9) value to a char (1) value that has the same binary encoding. In other words, it converts the number K to the Kth ASCII character (zero-origin). It works on non-ASCII characters as well, those > 177 octal.

The byte builtin is the inverse of the rank builtin, and has been added for similar reasons. The following typical old constructs should be changed to use byte:

```
OLD:    substr (collate9 (), i + 1, 1)
NEW:    byte (i)

OLD:    unspec (char1) = substr (addr (i) -> char4, 4, 1)
NEW:    char1 = byte (i)
```

The byte builtin is available in Release 25 and subsequent releases.

IMPROVED BUILTINS

The verify, ltrim, and rtrim builtins have been improved in the cases where the 2nd argument is a char(1) constant (either a literal constant, an options(constant), or substr of one of these). Note that this includes the common cases for ltrim and rtrim where the 2nd argument is omitted, and hence is a single blank. Formerly the compiler generated a 128 word table in the text section of each program that used verify, ltrim, or rtrim with a constant second argument. (An operator call was, and still is, generated for a variable second argument). Release 25a of PL/I (the MR8.0 compiler) generates a reference to one of 512 possible tables in pl1_operators_ for the char(1) constant cases.

This optimization means that every program that uses verify, ltrim, or rtrim with a constant, char(1) second argument will become 128 words shorter when compiled with release 25a. Further, since pl1_operators is wired, no page fault will ever be taken on these tables. Rather than attempting to determine whether a program that uses these builtins meets the optimization criteria, the best technique is simply to recompile all programs that use verify, ltrim, or rtrim. Obviously, no source changes are necessary.

I recommend that all programs that use verify, ltrim, rtrim be recompiled for MR9.0.

signed and unsigned ATTRIBUTES

The signed and unsigned attributes have been added to the language to enable the programmer to specify whether a real (i.e., not complex) variable should have storage for a sign, or not. These attributes specify the sign-type of a variable. The unsigned attributes specifies that a value represents nonnegative values only, and therefore no sign is needed nor wanted.

The present definition permits only real fixed binary variables to have the unsigned attribute. Someday we will permit real fixed decimal values to have it, too. Any variable declared with the fixed or float attribute can have the signed attribute. See the PL/I Language Specification (AG94), section 5.5, page 5-34, which explains attribute compatibility, for a complete description.

A value declared "fixed bin (18) unaligned" takes 19 bits. A value declared "fixed bin (18) unaligned unsigned" takes only 18 bits. Hence, the unsigned attribute can be used to "squeeze out" one more bit of storage, or conversely, to store twice as many values in the same storage.

The signed and unsigned attributes are defined (and implemented) as a property of variables in storage; it is meaningless to talk about the sign-type of an expression. As with the data-type and alignment attributes, the sign-type must match in a call-by-reference context.

The abbreviation for unsigned is uns. The signed attribute has no abbreviation.

The maximum precision for fixed binary values remains 71; there is no fixed bin (72) unsigned. The switch from single-precision computations to double-precision computations remains at 35/36; fixed bin (36) unsigned is double-precision.

Aligned unsigned variables differ from unaligned unsigned variables in two respects. First, only unaligned unsigned variables actually save storage. Aligned binary variables continue to take one or two words, as appropriate. Unaligned binary variables take an exact number of bits, as before. Second, the rule for whether one word or two words is used differs between aligned and unaligned unsigned variables. The split is at 35/36 for aligned, and at 36/37 for unaligned. Normally, this should not concern the programmer, since the normal PL/I precision and argument-matching rules ensure that the compiler can't get confused. The only time the programmer needs to think about the difference is when he or she is counting every word carefully.

It is invalid PL/I to assign a negative value to an unsigned variable. The size condition prefix, if enabled, detects this error. A value of zero is permitted.

The signed and unsigned attributes are available in Release 24 and subsequent releases.

PACKED DECIMAL

One of the major changes in Release 25 is the change from a 9-bit representation to a 4-bit representation for unaligned decimal variables. Thus, unaligned decimal variables take about half the space as aligned decimal variables. If P is the precision (number of digits), then the formulae are:

```
real fixed dec (P) unal:      (P+2)/2 bytes

real float dec (P) unal:      (P+4)/2 bytes

real fixed dec (P) aligned:   (P+4)/4 words

real float dec (P) aligned:   (P+5)/4 words
```

While decimal is infrequently used in systems programs, all of the programs that do use it, and have no compatibility problems, should be changed to use unaligned decimal. To suppress a warning message from the compiler, the "packed_decimal" option must also be given on the main procedure.

If you recompile a system program and suddenly get the warning about the change in the implementation of decimal, do NOT simply add the packed_decimal option without first ensuring that there are no compatibility considerations. The warning message is the only way we have of detecting programs that use decimal in the old way.