To: DISTRIBUTION

From: Steve Webber

Subject: New Command Processor Conventions

Date: 3/3/75

INTRODUCTION

This memo describes a proposed new calling sequence for command and active functions. The major change is that the command processor and the active function processor (proc_brackets_, today) will examine the entry descriptors of the command or active function about to be called and prepare an argument list appropriately.

Before describing the proposed new command calling sequences it should be noted that it is not being proposed that arbitrary argument lists be prepared or that any conversions be done by the command processor. Rather, the command processor will look for certain argument lists (that expect only character strings) and treat any that don't fall into this set the same way they are treated today, namely by calling them with the given number of char (*) unaligned arguments.

NEW COMMAND ARGUMENT LISTS

There are three basic formats of argument lists that the command processor will initially special case. These are:

```
     1.  command:     proc (args);

     2.  active func: proc (args, af_switch, ret_ptr);

 and 3.  command:     proc (arg1, ..., argN);
```

where:

1. args is an array of varying length character strings declared in one of the following ways (N and M are constants):

   i. (*) char (*) varying.

   ii. (N) char (*) varying.

   iii. (*) char (M) varying, and

   iv. (N) char (M) varying.

2. af_switch is a switch, set by the command processor or active function processor, indicating the entry was called as a command ("0"b) or as an active function ("1"b).

3. ret_ptr is a pointer set by an active function to point to the value of the active function. (See below.)

4. arg*i* are varying strings of fixed or variable maximum length.

A command writer chooses one of the above formats depending on whether the command can also be called as an active function. If the command can also be called as an active function or if the program can only be called as an active function, the second format is used. Otherwise, the first format will generally be used. The third format will be used by commands that always expect/require the same number of arguments The third form of program will never be called by the command processor (it cannot be used as an active function) unless exactly the correct number of arguments were given.

The args array can be declared by the command in the most appropriate way for the command. In particular, if the command must receive a given fixed number of arguments (and form 3 is not wanted) the command should declare args as

(N) char (*) varying, or

(N) char (M) varying.

If N arguments are not given on the command line, the command processor will not even call the command but will rather print an error message such as:

Incorrect number of arguments passed to <command name>.

This isolates such checking in the command processor so that each command need not do it.

If a command is willing to accept a variable number of arguments one of the following declarations for args should be used:

```
    (*) char (*) varying, or

    (*) char (M) varying.
```

When this is the case, the command can easily (and efficiently) find the number of arguments by using:

```
    hbound (args, 1)
```

Similarly, to reference the n'th argument one merely uses:

```
    args (n)
```

rather than a (more costly) call to cu_$arg_ptr or cu_$af_arg_ptr.

A further advantage is the ease with which a command that can also be used as an active function can be written. Arguments would be referenced in the same way regardless of how the program is being used.

Note that the fewer asterisks in the declaration of args, the faster will be the accessing code in the command program. If either

```
    (*) char (M) varying, or

    (N) char (M) varying
```

were specified and the command line gave arguments longer than M characters, the command processor would not call the command but would rather print a message such as:

```
            Argument <N> passed to <command name> is too long.
```

In many cases this relieves the command program from checking the length of its arguments.

RETURNING ACTIVE FUNCTION VALUES

A special entry in cu will be provided for active function use. The effect is a returns (char (*)[varying]) but done with fewer data copies and hence more efficiently. The use is:

```
  declare cu_$return_value entry options (variable);

     call cu_$return_value (value of active function);
```

This program extends the stack frame of its caller's caller and copies the given string into the extended region. (The parameter "value of active function" may be a varying or nonvarying string expression.) It also sets the third argument to its caller's caller (ret_ptr, above) to the first word of the extended region. The buffer contains a based, varying string -- hence the returned value is referenced as:

```
declare returned_value char (100000) varying based (ret_ptr);
```

The active function processor may release the storage allocated by cu_$return_value by a call to cu_$shrink_stack_frame.

CHANGES TO EXTERNAL INTERFACES

The above proposal makes the following entries in cu unnecessary:

cu_$arg_count

cu_$af_arg_count

cu_$ptr_call

cu_$arg_ptr

cu_$arg_ptr_rel

cu_$af_arg_ptr

cu_$af_return_arg

The new entry cu $return value must be provided. The cu_$ptr_call entry is necessary because of the inconvenience and inefficiency in converting pointer variables to entry variables in Multics PL/I and will certainly have to be retained for other reasons. (All entries will. of course. have to be retained forever.)

WHAT NEEDS TO BE DONE

The following tasks must be completed in order to complete the proposed changes:

1. The PL/I compiler must be changed to generate the (newly proposed) standard entry structures which make it more easy and efficient to find entry descriptors. The basic change in the entry definition is the movement of the entry descriptor pointers from the definition section to the text section.

2. The programs "command_processor_" and "proc_brackets_" must be changed to examine entry descriptors (if and only if the entries are of the new form) If entry descriptors are used (the command expects explicit arguments) appropriate action as mentioned above is taken.

   The command processor and active function processor will also be changed at this time to take appropriate action when an incorrect number or size of arguments is noticed.

3. As time permits, commands should be modified (probably as part of some other optimizing project) to use the new conventions. Full documentation must have been provided to allow command writers (and readers) to know what the conventions are.