To: Distribution

From: Robert A. Freiburghouse

Subject: The Command Language Revisited

Date: April 17, 1974

This document defines a command language and command processor that is intended to be a user selected alternative to the current Multics command processor. The language is suitable for use as an interactive or absentee job control language, and it also is a suitable language in which to perform simple calculations.

Design Objectives

1. Provide a single unified language containing the essential functions of calc, abbrev, do, exec_com, absentee and the current Multics command language.

2. Provide a command language that can call subroutines and functions written in standard languages in a natural manner passing arguments and receiving values having any of the scalar data types of the standard languages. Any procedure whose arguments and return values are scalars can be invoked from the command processor exactly as it would be invoked from another procedure, thus eliminating the need for active functions and commands to be written in a nonstandard style.

3. Provide a language whose implementation will perform a given operation using less CPU time and storage than used by the existing command processor and related facilities to perform the equivalent operation.

General Concepts

The command language is a very simple algorithmic language whose largest syntatic unit is a <command>. Each <command> is a conditional or unconditional imperative statement which can contain references to named variables, expressions, and other <command>s. Expressions are the familiar parenthesized infix and prefix expressions of Fortran or PL/I.

The command processor is an interpreter that executes a sequence of <command>s. Interpretation of each <command> is performed as a two stage process. During the first stage, the <command> is processed as a sequence of characters without regard to its syntatic construction or purpose as a command. It is during this first stage that abbreviations and parameters are replaced as described later. During the second stage of interpretation, each <command> is parsed (identified) and executed.

The command processor can be called by a <command>. Each invocation creates a new set of arguments, a new set of local variables, a new command input file, and a new "current" abbreviation file.

```
cp source s1 s2...sn
         or
cp source
         or
cp
```

where source is a pair of strings that identifies the command input file, and s1 s2...sn are strings which are arguments of the new invocation of the command processor. If source is omitted, commands are read from user_input. A more precise definition of the relationship between an invocation of the command processor and its I/O attachments is given in Appendix A.

Parameter Substitution and Abbreviation Replacement

Before each <command> is executed, the following steps are performed:

1. Each &K or &rK, where K is an <integer>, is replaced by the Kth argument to this invocation of the command processor. If no such argument exists, the &K or &rK is removed. The replaced text is rescanned from left-to-right. Parameters of the form &rK cause the replaced text to be quoted and any contained quotes to be doubled. This step is complete when the first ; or newline not contained in quotes is encountered.

2. If the string produced by step 1 begins with an !, the ! is removed and processing continues with step 3.

   If the current instance of the abbrev I/O switch is attached, the file is used as an abbreviation file. Any token defined by the abbreviation file is replaced by text from the abbreviation file. The replaced text is not rescanned.

3. Each occurrence of [<text>] is processed as if it were a command and the value of the command is used to replace the [<text>]. The value must be a character string. The processing of the <text> continues from this step, thus allowing nested instances of [<text>].

4. If the command does not begin with a ., perform the following steps:

   a. Parse the command as a sequence of <symbol>s s1 s2...sn using the following syntax:

      <symbol-list>::= <symbol>...

      <symbol>::= <quoted string>|<unquoted string>|
           <parenthesized string>

      <quoted string>::= "<char>..."

      <char>::= ""|Any ASCII character except"

      <parenthesized string>::|= (<char>...)
           such that parenthesis are balanced.

      <unquoted string>::= <not parenthesized> <notend>...

      <notend>::= Any ASCII character except blank,
           tab, newline, or ;

      <not parenthesized>::=
           any <notend> except left parenthesis

   b. remove the surrounding parenthesis from each <parenthesized string>

   c. Surround each <unquoted string> with quotes.

   d. if one or more sequences of <symbol>s is described by:

      {<symbol> [<symbol>]...}

   then all such sequences must have the same number of <symbol>s and must must not contain any nested occurences of such sequences.

   Let n be the number of <synbol>s in each such sequence. For k=1,2,...n, replace each sequence with its kth <symbol> and perform step e.

   If no such sequence exist, perform step e once.

   e. Rewrite the command as:

```
.call s1 (s2, s3, ..., sn)
```

Note, the actions performed for step 4 allow calls to be typed with a minimal syntax very similar to the syntax used by the current command processor. Braces cause iteration as do parentheses in the current command language. Parentheses are used to embed expressions into this type of command. Square brackets are used as active functions in any command line and are processed as part of the string processing that occurs prior to execution of the command as described in step 3. The special significance of [] {} () ! & and ; can be suppressed by use of the escape character ~.

Note that by using abbreviations the user can eliminate the . required on each command and can change the syntax of commands to a limited extent.

The Syntax and Semantics of Commands

```
<command>::= <attach>|<detach>|<do>|<exit>|
     <while>|<if>|<let>|<call>|<return>|<print>|
     <on>|<abort>|<for>

<attach >::= .attach <switch> <source>

<switch>::= command_input|user_input|abbrev

<source>::= switch <expression>|
       path <expression>|
       string <expression>
```

causes the attachment of the current instance of the <switch> to be "pushed down", and the <switch> to be attached to the <source>. If <source> is string <expression> the character string value of the <expression> serves as the file.

```
<detach>::= .detach <switch>
```

causes the attachment of the current instance of the <switch> to be "popped up", that is, replaced by the previous attachment of that <switch>.

```
<do>::= .do <group>

<group>::= <command>|(<command>[;<command>]...)
```

causes the <command>s of the <group> to be executed by the current invocation of the command processor. Normally a <do> is used as part of a compound <command> such as <if>, <while>, <for> or <on>.

<exit>::= .exit

causes the execution of the current <do> to be terminated and the <command> following the <do> to be executed. It is an error to execute an <exit> outside of a <do>.

<while>::= .while<expression><do>

If <expression> is true, the <do> is evaluated; otherwise, it is not. Upon completion of the <do>, the <while> is repeated. The <expression> must yield a logical value.

<if>::= .if<expression><do>

If <expression> is true, the <do> is evaluated; otherwise, it is not. The <expression> must yield a logical value.

<let>::= .let<name> be <expression>

causes <name> to be defined as a local variable allocated in the current stack frame. The value of the variable is the value produced by evaluation of the <expression>.

<call>::=
.call <expression>([<expression>[,<expression>]...])

Evaluation of the first <expression> must yield a string giving a pathname that identifies an object segment entry point.

The argument <expression>s are evaluated and converted to conform to the data types specified by the entry definition of the object segment as described later.

<return>::= .return

causes control to return from the current invocation of the command processor.

<print>::= .print <expression>[,<expression>]...

causes the value of each <expression> to be written on user_output in a suitable format.

<on>::= .on <expression> <do>

causes the <do> to be established as an on-unit for the condition identified by the string value of the <expression>. The <expression> must yield a string value. The execution of an <exit> or the normal termination of the <do> causes control to return to the signaller. If control is to be returned to the <command> following the <command> whose execution caused the signal, an <abort> must be executed.

<abort>::= .abort

causes execution of the <command> following the <command> whose execution caused the most recent signal. It is an error to execute an <abort> not continued within a <do>, used as an on-unit.

```
<for>::= .for <name>=<expression>[,<expression>]...<do>
```

Let n be the number of <expression>s. For k=1,2,...n, the kth <expression> is evaluated and its resulting value assigned to the local variable <name>, and the <do> is evaluated. A <for> defines its <name> as a local variable just like a <let>.

```
<name>::= .<identifier>

<identifier>::= <letter>[<letter>|<digit>|=]...

<expression>::= <infix>|<prefix>|<basic>

<infix>::= <expression><infix-op><expression>

<infix-op>::= +|-|*|/|**|= |^= |>= |<= |<|>|&|||||

<prefix>::= {+|-|^}<expression>

<basic>::= (<expresison>)|<name>|<constant>|<function>

<constant>::= <identifier>|<quoted string>|<integer>
              |<real>|true|false|null

<integer>::= <digit>...

<real>::=
     {<integer>.[<integer>]|.<integer>}[e[+|-]<integer>]

<function>::= <expression>([<expression>[,<expression>]...])
```

A function works like a call, except that a return value is expected and is converted to the corresponding command language data type.

Variables and Data Types

A local variable is allocated in the stack frame of the command processor. Each variable is capable of possessing values of any data type.

The possible data types are:

integer	(fixed bin(35))
real	(float dec(18))
logical	(bit(1))
string	(char(256) varying)
address	(pointer, pointer)

These data types are designed to accommodate all PL/I and Fortran data types except complex numbers. The conversions between these types and PL/I types are given in the following section.

A variable is defined by the appearance of its <name> in a <let> or <for>. Because the command language has no concept of multiple scopes of names and no declared attributes, no declarative statements are required. The type of a variable is the type of the value it currently possesses.

Argument Conversion

If an entry definition specifies no parameters, the arguments, if any, are passed without conversion.

If the entry definition specifies a single one-dimensional array, the arguments are converted to the data type of the array and each argument is transformed into an element of the array. The lower bound of the array descriptor is set to 1 and the upper bound is set to n, where n is the number of arguments given. Using this scheme, a PL/I procedure can easily receive a variable number of arguments while remaining within the standard language.

If the entry definition specifies one or more scalar arguments, each argument is converted to the data type of its corresponding parameter. If an argument is a reference to a local variable, it is passed by-reference; otherwise, it is passed by-value. When an argument is passed by-reference, it is converted to conform to the data type of the corresponding parameter, and upon return it is converted back to the original type of the argument.

If the expected data type of a called procedure is any kind of PL/I arithmetic data, both integer and real can be converted to the expected type. On return, all PL/I arithmetic types, except complex, can be converted either to integer or real. Large decimal values are rounded and a warning produced.

Aggregate values cannot be passed or received.

PL/I bit strings, other than bit(1), are converted to character strings.

Excessively long (>256) character strings are truncated with a warning.

Because the command language stores addresses as pointer pairs, it can hold pointer, offset, label, entry, format, file, and area values as address values.

Appendix A

Each invocation of the command processor establishes a new instance of three I/O switches: command_input, user_input, and abbrev. These three switches are attached in the following manner.

command_input:

If this invocation of the command processor received two or more arguments, the first two arguments identify the file to which command_input is attached.

If any arguments are given, at least two must be given. The first two arguments define how command_input is to be attached. The attachment is exactly the same as that specified for the <attach> command on page 4.

If no arguments are given, command_input is attached as a synonym for the previous instance of command_input. If no previous instance exists, it is attached as a synonym for user_input.

user_input:

user_input is attached as a synonym for the previous instance of user_input. If this is the first instance, it is attached as a synonym for user_io.

abbrev:

abbrev is attached as a synonym for the previous instance of abbrev. If this is the first instance it is not attached.

Appendix B

Changes in this revision:

1. iteration is allowed when mapping from an unpunctuated call to a <call>.

2. [<expression>] has been replaced by [<text>]. Providing a more compatible treatment of active functions.

3. the command processor can be invoked with the set of commands it is to execute, providing an equivalent of the current "do" command.

   Examples:

```
                                          rename a x
rename {a b c} {x y z}    rename b y
                                           rename c z

cp string "pl1 &1;dp &1.list" foo    do "pl1 &1;dp &1.list" foo
                                          (in existing system)

[fun a b]>x y    [.call "fun"("a","b")]>x y
     result>x y    .call "result>x" ("y")
```