TO: MSPM Distribution
FROM: K. Martin
SUBJ: BX.2.00, The Shell
DATE: May 26, 1967

This revision of BX.2.00 incorporates four major changes:

1) The concept of context commands has been added.

2) Housekeeping is done by the listener procedure (BX.2.02).

3) An argument to the Shell has been added, indicating whether or not pathnames should be accepted as command names.

4) The listener procedure is now in the working process. Consequently, the Shell uses the standard error handling mechanism rather than an error argument.

Published: 05/26/67
(Supersedes: BX.2.00, 09/10/66;
BX.2.00, 10/21/65)

The Shell

R. Sobecki, G. Schroeder, K. Martin

Purpose

The Shell is the procedure called by any other procedure that wishes to have an ASCII string interpreted, according to the rules of the Multics command language, as a command sequence. Command sequences are fully described in MSPM BX.1.00, Multics Command Language; briefly, a command sequence is a command or group of commands that the Shell is expected to execute.

Introduction

The Shell is called by

```
call shell (comstring, reject_pathnames);
```

comstring is an ASCII string which is to be interpreted as a command sequence.

reject_pathnames is an indicator to the Shell of whether it should allow pathnames as commands ("0"b) or restrict commands to entry names ("1"b).

The PL/I declaration for the arguments to Shell is

```
shell: proc (comstring, reject_pathnames)
       recursive; dcl comstring char (*),
       reject_pathnames bit (1);
```

MSPM BX.0.00, Overview: Use of Commands in Multics, describes the Shell's role in the command system. A brief review might be useful here.

The listener procedure reads messages from the user's command input device (usually a teletype or typewriter console). Any message read by the listener is expected to be a command sequence. Command sequence is defined as several queued commands separated by semicolons, and finally, a command terminated by a new line character.

After the listener has read a line from the command input device, it scans it to see if it should read another line to complete the command sequence. The listener looks for two things to indicate that it must read another line:

1) a new line character <NL> immediately preceded by the Shell escape character (%).

2) a <NL> imbedded in an imcomplete literal string (a literal string is enclosed by `').

When the listener has a complete command sequence it calls the Shell with the command sequence as the first argument.

The Shell interprets the ASCII string and calls the appropriate commands. When the last command in the sequence returns, the Shell returns to the listener. The user is now at command level and so may issue another command sequence.

Implementation

The Shell consists of several procedures. At the outermost level is the driver procedure. The driver is responsible for error handling. The driver calls the Command Sequence Interpreter and Initiator (CSII) to scan the ASCII string and initiate the command. CSII consists of two phases. The first phase scans a command sequence and sets up threaded lists representing a syntactic analysis of the entire command sequence to be executed. The second phase performs the command sequence execution. In the second phase, CSII sets up linkage to and calls the commands in sequence with argument lists. The two phases are necessary to avoid command execution before the command sequence has been checked for syntax errors.

CSII contains a number of procedures which accomplish the syntax analysis of the commands in the command sequence. They are:

1) scan, which moves the scanning pointer across the command string;

2) comlist, which interprets commands and lists within commands;

3) elementi, which interprets and classifies elements within commands (see MSPM BX.1.00 for a definition of element);

4) separatori, which interprets separators within commands (see MSPM BX.1.00 for a definition of separator).

Another procedure within CSII, comex, builds argument lists from the results of the analysis of the above procedures. Comex also calls the commands in proper order.

Figure 1 shows how these procedures fit together.

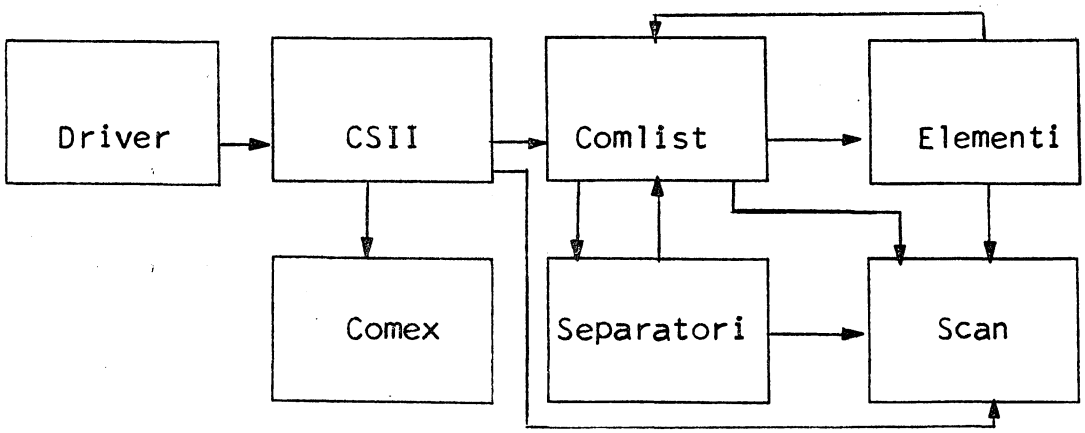

Figure 1

Error Handling in the Shell

The driver procedure in the Shell handles any errors which occur during the execution of the scanning of the ASCII string and the attempt to call the required command. The driver "enables" for all defined errors by making a series of on condition (error name) statements. When an error occurs at any level of nesting, control returns to the outer level of the Shell, the Driver, by use of signal condition (error name) statements. The driver then generates a message for the output stream giving the nature of the error, sets the error indicator and returns to the Shell's caller.

The Shell also handles errors which are encountered during the execution of a command sequence if these errors are not specifically provided for within the procedures included in the command sequence. When a programmer-defined condition is signaled for which no procedure currently in the stack has issued an "on condition" statement, the standard system action is to call a library procedure, unclaimed_signal (BY.11.05). Unclaimed_signal prints a message from the information in the error file (produced by the error handling procedures in the library; see BY.11.00 - BY.11.04), then gives the user a choice of continuing or issuing a new command. If the user chooses to continue, unclaimed_signal returns to the procedure that signalled the error. A new command may be unrelated or may wish to investigate the error; these two possibilities are handled as discussed in BY.11.05. When the Shell regains control it signals an error to its original caller. Usually, the Shell's caller is the listener (see BX.2.02), which does not care that the Shell intercepted an error. However, other procedures, such as the debugging aids, may call the Shell and be vitally interested that an error occurred. The Shell signals the error for the benefit of these other procedures.

The Threaded Lists Representing the Command Sequence

The Shell's procedures set up the command sequence as a threaded list of data structures in controlled storage. The list consists of a thread of data structures representing a command name, the arguments to a command, and any interjected commands included within the command. The name of a command may be a string, a literal string, or an immediate-value command. An argument to a command can be a string, literal string, immediate-value command or a one-dimensional array (list) of strings, literal strings, or immediate-value commands. Interjected commands may be interspersed with the elements of an array. Each data structure in the threaded list may represent a string, literal string, immediate-value command, interjected command or list.

The command sequence may consist of several queued commands separated by semicolons, and finally, a command terminated by a new line character. The command sequence itself is set up as a threaded list of PL/I data structures. Each of the structures in the command line thread represents one queued command on the line. The following declarations describe the threaded list:

```
dcl root_ptr ptr auto;

dcl 1 element_description ctl (p),

    2 next_ptr ptr,
    2 data_ptr ptr,
    2 etype bit (4), /* 0001 = string
                        0010 = literal string
                        0011 = immediate-value command
                        0100 = interjected command
                        0101 = list */

    2 ecount fixed;

dcl 1 element ctl (q),

    2 size bit (9),
    2 data char (q->element.size);
```

Root_ptr defines the beginning of the thread. Element_ description defines an element in the command sequence.

The first pointer (next_ptr) in element_description points to the next data structure. Next_ptr is null if this is the last data structure in the thread. The second pointer (data_ptr) points to the data if etype is string or literal string, or to another thread if etype indicates immediate-value command, list, or interjected command.

Ecount is null for string or literal string. Ecount contains the count of elements in a list or command for the other three types. (Count of elements in a command equals count of number of arguments plus one for the command name.) Interjected commands within either commands or lists are not included in the count of elements. The commands in the command sequence are represented with etype = 0011 (immediate-value command) by convention.

Figure 2 illustrates a thread resulting from the syntactic analysis of the command sequence:

add {typein [printstring `type argument one']} arg2 <NL>

Analysis of the Command Sequence

The Command Sequence Interpreter and Initiator (CSII) is the controlling procedure for the procedures which anlayze the command sequence and build the thread of element_ description structures and for the procedures which execute commands.

CSII calls comlist to analyze the first command in the sequence. Upon return from comlist, CSII checks the current scan character. If it is ";", CSII calls comlist to analyze the next command, etc. When, upon return from comlist, CSII finds that the current scan character is <NL>, CSII calls comex to execute the command sequence.

CSII threads together the element_description structures returned by comlist. Each structure in the thread represents a command in the sequence (see example, p.7).

Comlist

Comlist is called to analyze a string which is expected to be either a command or a list. Comlist is called by CSII for each command in the sequence; it is called by elementi when an element is found that is an immediate-value command or a list; and it is called by separatori when an interjected command occurs in a separator. (A command language separator is a combination of spaces, comments, and interjected commands; see MSPM BX.1.00 for a formal definition).

If comlist is analyzing a command, it removes all labels before the command. If it is analyzing a list, it does not look for labels.

Comlist calls elementi to get each element of the command and calls separatori whenever a separator is expected in the command, i.e, according to the following definitions (given in augmented Backus normal form - See MSPM BX.1.00):

<labeled command>::=<<label> <separator>>0 <command>

<command>::=<command name> <<separator> <element list>>1-

<element list>::=<element> <<separator> <element>>0

When elementi returns after operating on a command name, comlist (if pathnames are not allowed as commands) checks the command name for the "greater than" character (>) or the "less than" character (<). If either ">" or "<" is found, the command name is a pathname and is illegal. Comlist signals an error to the driver. Optional restriction of command names to entry names makes possible a class of restricted users who may not specify commands by pathname.

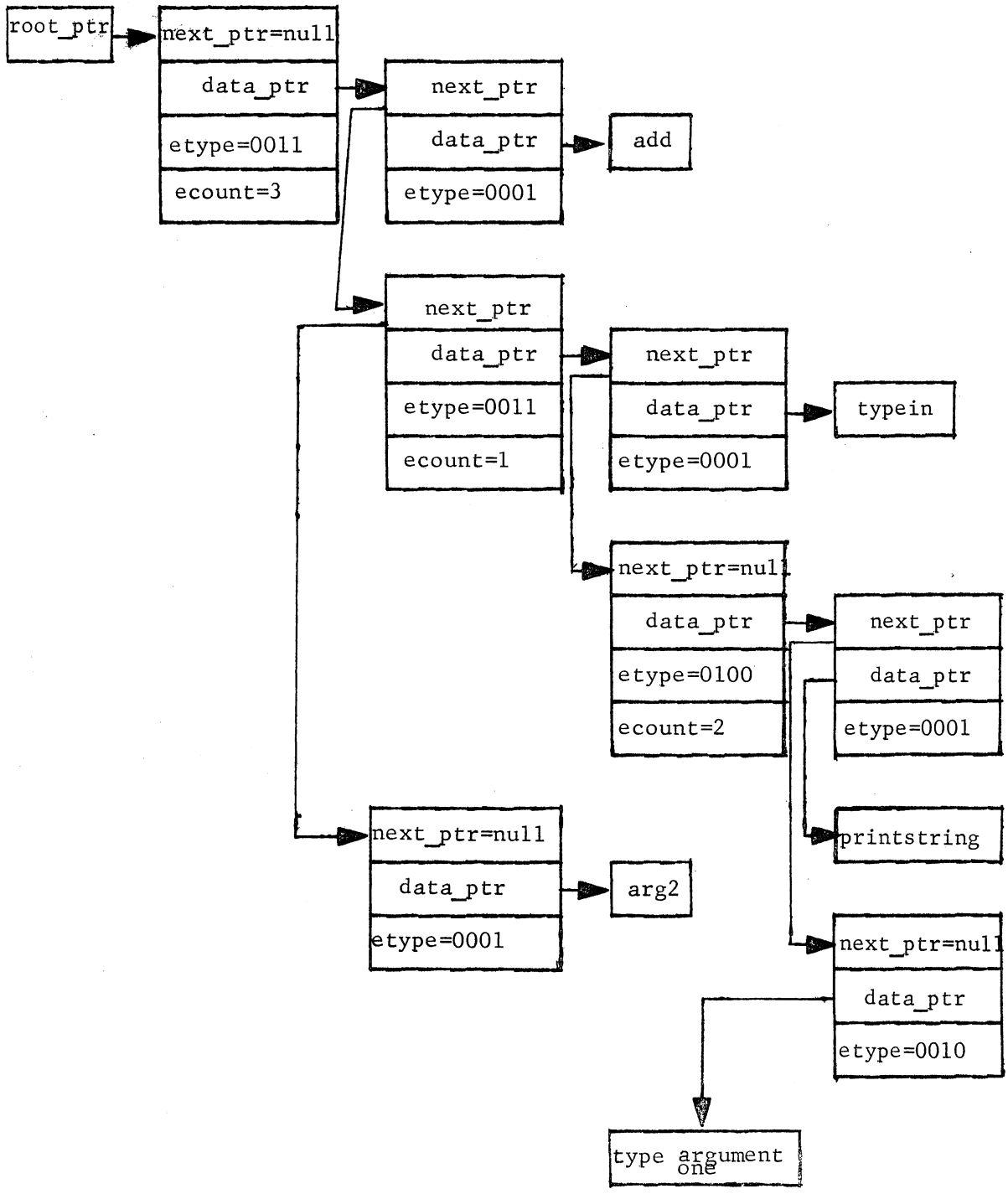
root_ptr
next_ptr=null
data_ptr
etype=0011
ecount=3
next_ptr
data_ptr
etype=0001
add
next_ptr
data_ptr
etype=0011
ecount=1
next_ptr
data_ptr
etype=0001
typein
next_ptr=null
data_ptr
etype=0100
ecount=2
next_ptr
data_ptr
etype=0001
printstring
next_ptr=null
data_ptr
etype=0001
arg2
next_ptr=null
data_ptr
etype=0010
type argument one

Figure 2

When either elementi or separatori gives a return to comlist indicating that no element or no separator was found, comlist returns to its caller. Comlist constructs a thread of element_description structures representing the elements of the command and any interjected commands found within the command and returns a pointer to this thread to its caller.

The PL/I declaration for the arguments to comlist is

```
comlist: proc (eletype, lblind, cl_ptr) recursive;

dcl eletype bit(4), /* 0011 = immediate-value command,
                       0100 = interjected command,
                       0101 = list */

    lblind bit (1),/* 0 = no labels 1 = check for labels*/

    cl_ptr ptr;/* returned by comlist; pointer to structure
                  representing command or list found */
```

Elementi

Elementi is called by comlist to find and classify elements in the command or list being built by comlist. Elementi determines which type of element it has by noting punctuation and builds a proper element_description structure. Elementi is called with three arguments. The following is a PL/I declaration for the arguments.

```
elementi: proc (lstind, struct_ptr, no_elem) recursive;

    dcl lstind bit(1),      /* 1 = list allowed as element
                               0 = no lists allowed */

        struct_ptr,         /* returned by elementi;
                               a pointer to structure
                               representing element found */

        no_elem bit(1),     /* returned by elementi;
                               0 = element found;
                               1 = no element found */
```

Separatori

Separatori is called by comlist to find and skip over separators in a command. Separatori skips over spaces and comments and looks for interjected commands. When separatori discovers an interjected command, it calls comlist to analyze the command. Separatori threads together all interjected commands found in a single separator and returns a pointer to this thread to its caller.

The PL/I declaration for the arguments to separatori is:

```
separatori: proc (lsind, beg_ptr, end_ptr, no_sep) recursive;

     dcl lsind bit(1),          /* 0 = ordinary separator expected
                                   1 = left space type separator
                                   expected */

     beg_ptr ptr,               /* returned by separatori;
                                   pointer to beginning of thread
                                   of any interjected commands
                                   found */

     end_ptr ptr,               /* returned by separatori;
                                   pointer to last structure in
                                   thread of interjected commands */

     no_sep bit(1);             /* returned by separatori;
                                   0 = separator found
                                   1 = no separator found */
```

Scan

Elementi and separatori rely heavily on the scan module. The scan module does the scanning of the command input string.

The scan module has four entries to accomplish various scans:

1) Scan_skip skips one character;
2) Scan_string scans to the end of a string;
3) Scan_litstr scans to the end of a literal string;
4) Scan_cmnt skips to the end of a comment.

Scan also recognizes and treats the Shell escape character.

The arguments to all the entries are the same. Scan_skip is given as an example.

```
scan_skip: entry (cstring, scsubstr, scptr, scind);

  dcl cstring char(*),          /* The command string */

  scsubstr char (*) varying,    /* Set by scan_string and
                                   scan_itstr to contain string
                                   or literal string found;
                                   scan_skip and scan_cmnt leave
                                   scsubstr unchanged */

  scptr fixed,                  /* index of current scan
                                   character in cstring */

  scind char (1);               /* contains current scan
                                   character if it is punctuation;
                                   otherwise, null */
```

Executing the Commands

CSII initiates its command execution phase as soon as it encounters in the command sequence a <NL> which is not included in a literal string nor immediately preceded by a Shell escape character. CSII examines the data-ptr in the element_description pointed to by root_ptr; if data_ptr is non-null, CSII calls comex with a pointer to element_description. When comex returns, CSII determines whether there is another element_description threaded to the one just operated on by comex; if there is another, CSII again calls comex with a pointer to that element description; and so on until CSII comes to the end of the thread. CSII then returns to its caller.

Comex

The PL/I declaration for the arguments to comex is:

```
comex: proc (c_ptr, c_value) recursive;

       dcl c_ptr ptr,          /* pointer to first structure in
                                  a thread representing command to
                                  be executed */

           c_value ptr;        /* returned by comex; pointer to
                                  value returned by command execute*/
```

Comex examines the structure pointed to by c_ptr. It calls a program which pushes down the user's option data base if the command is an immediate-value command. The first element_description structure in the thread which has etype equal to string, literal string or immediate-value command is interpreted as the command name. If etype for this element_description indicates an immediate-value command, comex calls itself to evaluate the immediate-value command. The value returned by comex is assumed to be a string representing the command name. The command name is scanned for "$"; the substring to the left of "$" is assumed to be a segment name and the substring to the right of "$" is assumed to be an entry name within that segment. If no "$" is found in the command name, the command name is presumed to be both the name of the required segment and the required entry point. Comex calls link_change$make_link (BY.13.03) with the segment name and entry point name representing the command to be called. The procedure link_change$make_link returns a pointer to the linkage fault it has just built representing the command to be called. Comex sets up the argument list for a standard call (with all arguments as varying strings), building appropriate data specifiers and dope vectors.

If a list is encountered, the elements are set up as an array of varying character strings and appropriate data specifiers and dope vectors are built. If an immediate-value command is encountered as an argument or as an element of a list, comex calls itself. Upon return, it obtains the value pointer, c_value, and stores it in the argument list if the immediate-value command was a command argument. If the immediate-value command occurred as an element of a list, the value returned by comex is interpreted as a character string. The string is added to the array of strings. This process continues until all of the arguments have been set up. Space for one additional argument pointer is always allocated. This is the argument pointer in which the pointer to the value of the command is stored. When the argument list is complete, comex calls the command requested using the pointer to the linage fault returned by linkmk. The pointer to the value returned by the command is stored in c_value and comex returns to its caller.

Context Commands

Sometimes it is useful to think of an interjected or immediate-value command as being called in the context of another command. The command name in the immediately preceding nesting level of the command sequence is defined as the context command. Interjected commands which set global options (see BX.12.00) are an example of commands that need to know the name of their context command. In particular, the interjected command, brief, which sets the brief option may be called within the command, alpha. The briefcommand wants to know the command name, alpha, in order to set the local option, alpha.brief, as well as the global option, brief. Both the local and global options are set so that alpha is sure to find brief properly set and other procedures (which do not check alpha.brief) find the option set globally.

When an interjected or immediate-value command occurs as the first element in a nesting level, its context command is null. No command name has appeared at that immediately preceding level to be defined as the context command. The case of an interjected or immediate-value command as first element of a command is a special case of the above situation.

Implementation of the context command facility is by means of the signaling mechanism of PL/I. The procedure CSII executes an "on condition" statement defining the current context command before calling the procedure comex, and comex executes an "on condition" statement shortly after being called. The recursive nature of comex results in a stacking of on conditions corresponding to the nesting in a command sequence.

When a command wants to know the name of its context command, it merely signals the defined condition. The "on unit" sets an external static pointer to the appropriate value--the null pointer if the context command is null; a pointer to the name of the context command if it does exist.

Immediately before calling a command at a given nesting level, comex executes a "revert" statement, causing the "on condition" of the previous comex (that is, for the immediately preceding nesting level) to be in effect. The "revert" in the first level of comex causes the "on condition" in CSII to be in effect.