TO: MSPM Distribution
FROM: K. J. Martin
DATE: 11/08/67
SUBJ: BX.18.00

BX.18.00 is essentially a re-issue of BX.1.01. No important changes have occurred beyond minor updating to conform with other parts of Multics. The section change was made in order to group the large number of new macro subsections under a single heading.

Published: 11/08/67
(Supersedes: BX.1.01, 01/09/67)

Identification

Macro Facility for Command Language
G. Schroeder, D. B. Wagner, K. J. Martin

Reference

The user must be familiar with BX.1.00 Multics Command Language.

Introduction

When using an on-line system, one usually finds that there are certain sequences of commands that one issues over and over with only minor changes in arguments. The ability to define such command sequences as macros, then to invoke such a macro by typing the macro name and a list of arguments to be substituted into the command sequence, provides two major conveniences:

1) economy of typing

2) economy of thought; i.e., a complicated sequence of commands need be thought out only once; once defined, the sequence can be considered as a whole.

Perhaps the more obvious feature which must be included in a macro facility for a command language is the ability to declare certain arguments within the body of the macro to be bound variables and to substitute for these bound variables at the time the macro is invoked. Macros of commands should be definable such that the user can substitute the arguments to commands within the macro and for names of commands within the macro.

Another obviously useful feature is the ability to execute commands conditionally within a macro. One can easily envision a macro such that the failure of one command within it would cause the user to wish not to execute the rest of the sequence. It should be possible to express such a condition and have the macro behave properly when invoked.

The Control Commands

The above features are provided through the "control statements" of the macro package. These "control statements" are actually commands. Obviously the number and complexity of control statements can - and will - grow, but the following is a summary of a basic set:

    macro_arg(a b c ...)

(BX.18.03) specifies names of the bound variables in the text following. Macro_arg causes all occurrences of the strings a, b, c etc., which stand alone to be replaced by the corresponding arguments given in the macro call. A string is said to stand alone when it is delimited by ASCII characters that are not alphabetic or numeric or the underscore (_).

    create_subst(a b c ...)

(BX.18.03) causes special symbols to be created and substituted for a, b, c, etc., in the text following. Created symbols are produced by concatenating the string "crs_" with a string obtained using the "unique identifier generator" described in BY.15.01. They are always distinct from each other and have a very good chance of being distinct from other file names, variables, etc. which are used in the macro.

Use of created symbols makes it possible to freely use temporary variables and segments with names which are irrelevant outside a given macro, without worrying about naming conflicts.

The sequence

    iterate x list `command sequence'

(BX.18.04) causes the text included in command sequence to be repeated several times over, once for each element of list. On each repetition the corresponding element of list is substituted for every occurrence of the string x standing alone as an element of a command. Iterate's can of course be nested to any depth.

The command

    conditional x `command sequence'

(BX.18.05) causes the specified command sequence to be a part of the macro expansion only if the string x is not a string of zeros (x is normally the value of an immediate value command such as value, see below).

Two commands which will be useful in conjunction with the <u>conditional</u> command are <u>setvalue</u> and <u>value</u>, which maintain a data-base of variable-names and associated values.

    setvalue x string

(BX.18.06) establishes <u>x</u> as the name of a variable with value given by <u>string</u>. Then the immediate value command

    {value `expression'}

(BX.18.06) evaluates the arithmetic expression given, which may include variables established using the <u>setvalue</u> command, and returns a character-string representation of the value. In particular the command

    conditional {value `a < b + c'} `line'

(BX.18.05) causes the line to be included in the macro expansion if and only if the values of the variables <u>a</u>, <u>b</u>, <u>c</u> satisfy the expression a<b+c. Rules for evaluating expressions such as a<b+c will be established later.

It might sometimes be desirable to switch back and forth between sources of input - sometimes the macro file should be the source, then sometimes another stream (probably the one connected with the typewriter). This can be done by an additional control command:

    include stream_name

(BX.18.07). This command causes the macro to read stream_name for successive lines instead of taking items from a data base of commands to be executed. Another occurrence of the include control command causes the first stream to be pushed down in a stack of input sources.

The control command

    end_include

(BX.18.07) causes the macro to again take items from its previous source.

An end-of-stream return on a call to read stream_name also causes the macro to revert to taking items from its previous source.

Using the Macro Facility

A user who wishes to make a sequence of commands uses the context editor (BX.9.01) to create a segment. This segment contains the commands which make up the macro, including regular commands, macro control commands, and user procedures.

A macro is defined by issuing the command

```
          macro macro_name
```

(described in BX.18.02) where macro_name is the name of the segment which contains a series of commands.

Macro writes a linkage section for the segment being defined as a macro. This linkage section defines the symbol which is the name of the macro with a special class number. The macro segment itself is not altered.

The macro is invoked by typing

```
          macro_name(arg1 arg2 ...)
```

where macro_name is the name of a file containing a macro which the user has previously defined with the macro command.

When the Shell attempts to build linkage to this macro the linker returns the class number of the symbol. The Shell thereby knows that a macro is being invoked. The Shell then calls the macro processor with command name (macro_name) and the array of arguments (arg1 arg2 ...). As usual, the Shell executes any interjected commands itself and does not pass them to the macro processor; in fact the Shell does a full syntax analysis and interpretation of the command line and passes the arguments to the macro processor.

Overview of Implementation

The macro processor sets up two data bases:

1) commands to be executed
2) a key to string substitutions to be made

These two data bases are shared with a procedure called the request handler (see BY.4.01), which actually performs string substitutions on demand and which also does various manipulations of these two data bases at the behest of the control commands.

The macro processor also performs a variety of initialization tasks such as attaching the stream user_input to the request handler (BY.4.01) so that a call to read will invoke the Request Handler, and turning off the housekeeping option to prevent the listener (soon to be called) from returning. The macro processor then calls the listener (BX.2.02) which reads from the stream user_input and calls the Shell with the string returned by the request handler. The command which the Shell then calls may be an ordinary command, or it may be a macro control command, or it may be another macro invocation. The Shell and the listener never know the difference; the macro control command simply calls the request handler (at the appropriate entry) to adjust its data bases. When the request handler returns, the macro control command returns to the Shell which returns to the listener. The listener again calls read (invoking the request handler) to get the next command, etc., until there are no more commands in the macro to be executed. A command inserted in the request queue by the macro processor then turns on the housekeeping option which causes the listener to return.

Figure 1 shows how the Shell, the macro processor, the listener, and the request handler interact. The first data base (Q in the figure) contains a stack of "bunches" of commands. Each bunch in this stack is a list of commands - probably the text of one macro.

The second data base (Sub in the figure) contains a key to the string substitutions to be done in the macro. This is also a stack which is pushed down on each occurrence of a new macro. The macro_arg control command maintains this data base from the macro_arg statements and the list of arguments given when the macro was invoked. The create_subst control command also contributes to maintaining this data base.

There are certain commands such as the debugging commands and the context editor which are highly interactive. When executing, these commands accept requests from the user which are essentially subcommands to the command in control. The ability to include requests to a command within a macro appears to be a desirable feature. Although on the one hand, it is desirable to <u>be able to</u> include the requests to an interactive procedure in a macro, it is undesirable <u>to be required</u> to include the requests to an interactive procedure in a macro.

In the normal, undiverted flow of a macro all input comes from the macro segment. Some special information must appear within the macro to change input sources - such as the include control command. The user may use interjected commands to accomplish this. For example, a command line within a macro might be:

```
          edit alpha [include stream1]; end_include
```

The Shell interprets and calls the interjected command (here, the include control command) before calling edit. Thus, when edit is invoked and requests input, the request handler reads from stream1. When edit is completed (and returns to the Shell) the next command executed is the end_include control command.

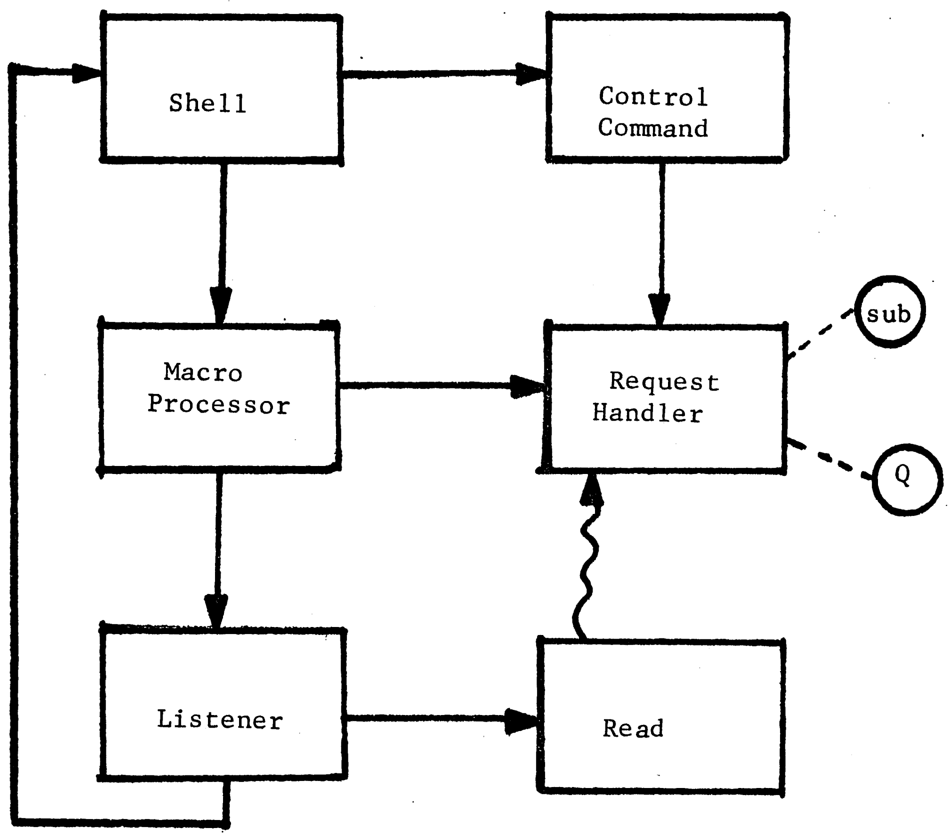
Shell
Control Command
Macro Processor
Request Handler
sub
Q
Listener
Read

Figure 1.