Wiley's Copy

SIMPLE LANGUAGE SYSTEM
TERMINAL COMMANDS
REFERENCE MANUAL

Mark Greenberg

May 1975

Systems Group

Technical Document

Center for Research in Management Science

University of California

Berkeley

This work was done as part of the Systems Development effort under National Science Foundation Grants GS-32138 and SOC75-08177-Balderston.

1.0 Introduction

The SIMPLE programming-language system provides facilities for creating, editing, compiling, executing, and debugging programs written in the programming language SIMPLE. The text editing facilities are generalized and therefore can be used to create and edit any textual store of information, not just SIMPLE programs.

This reference manual describes how to use the SIMPLE language system, giving details of all the commands that can be typed to the system from a terminal. Users interested only in general text editing should not read beyond Section 4.2 (text-editing commands) in this manual. Subsequent sections give details of commands used to compile, execute, and debug SIMPLE programs.

Other related systems-group technical documents are:

LINE COLLECTOR REFERENCE MANUAL
SIMPLE LANGUAGE SPECIFICATION REFERENCE MANUAL
INSTRUCTION PROCESSING UNIT FOR THE SIMPLE OBJECT LANGUAGE

This SIMPLE language system was designed and implemented by Charles Grant, Mark Greenberg, and David Redell.

2.0 Program Organization

At any time, the SIMPLE language system is working on a single program with a name specified by the user. The information associated with the program is organized into three segments (A segment is a linear sequence of data which can be identified by name.):

1) text segment - contains text of a program

2) code segment - contains compiled code of program

3) debugger segment - contains symbol tables and other information required to provide debugging features

The code and debugger segments are created only when the program is first compiled. Thus, when the SIMPLE language system is used only for text editing, a program consists of a text segment only.

A program is divided into blocks (which are stored sequentially within the segments). Block One is called the global block and contains the global declarations of a SIMPLE program. The first function of the program is in block Two, the third in block Four, and so on. (By convention, the function in block Two is the driver function of a program; it is the function called when a program is executed by the operating system as a subsystem. It should have no arguments.)

Each block of text is organized into a sequence of one or more lines. A line is a string of characters (not more than 128) ending with a carriage-return character. Thus, a program is just a sequence of lines of text which are divided into groups called blocks.

3.0 Command Structure

When the SIMPLE language system is entered, a colon is printed--indicating that the system is waiting for the next command to be typed by the user. A command is a string of characters terminated by a carriage return. Upon typing the carriage return, the typed-in command is interpreted and the indicated action is taken. When the command is completed another colon is printed, indicating another command may be typed. The user may type ahead although echoing of typed characters may be delayed in order to insure that output and echoed input are printed in the correct order. If the user types ahead too far, a bell will sound for each typed character ignored by the system. All line-collector features may be used during entry of a command line. The old line is the previous command line.

A command has the form of a command specification, possibly followed by one or more arguments separated by blanks. The command specification is the first two characters in the command line. They indicate which command is to be performed. Multiple blanks may occur wherever a single blank is legal. Blanks at the end of a command line are ignored. Blanks separating the first argument from the two-character specification are optional. If the command line begins with a blank, it is an immediate statement (described later).

Several commands separated by semicolons are allowed in one command line. If a command fails, the remaining commands in that line are not performed.

Examples of commands are:

```
:SP PROG                SP command with one argument
:LT "ABC" 2             LT command with two arguments
:CP                     CP command with no arguments
: F()                   immediate statement (starts with blank)
:LT "ABC";WH;ML         three commands separated by semicolons
```

Many commands will have arguments which specify a text-line address or a range of text lines. These will be described in detail.

3.1 Addresses

Addresses are used by the SIMPLE language system commands to locate text within a program. An address uniquely identifies a line of text within a program by specifying a block and a line within a block. An address consists of a string of characters which are interpreted by the SIMPLE language system. At any time, there is one line of text within a program which is singled out as the current line. In general, the current line is the last line which had some action performed on it.

In ALL commands that take an address as an argument, if the argument is omitted then the current line is assumed by default.

The first line of a block can be addressed by name or by number enclosed in slashes. For example,

:LT /3/	addresses block 3, line 0
:IT F3	addresses line Ø of block with function name F3

The name for block One is GLOBAL. Function names for blocks are legal only if the program has been compiled and the names have not been made invalid by subsequent creation or destruction of blocks.

Lines can be addressed relative to the beginning of a block. For example,

:AT /5/+7	addresses block 5, line 7
:ML F3+8-3	addresses the line three lines before the eighth line (i.e., line 5) of block with name F3.
:DT /2/3	addresses block 2, line 3. If the plus sign is omitted, it is equivalent to /2/+3.

Lines can be addressed relative to the current line. For example,

:EL 1	addresses the line after the current line
:LT -3	addresses the line three before the current line
:AT 1-2	addresses line before the current line
:AT +1	addresses line after the current line

Lines can be addressed by their contents. For example,

:LT /5/+"ABC"	addresses the first line that contains ABC after the first line of block 5. If line Ø and line 1 of block 5 both contain ABC then line 1 is addressed, since the "context search" starts after the previously addressed line.
:DT "XYZ"	addresses the first line after the current line that contains XYZ
:CT -"PQ"	addresses the first line before the current line that contains PQ

Lines can be addressed by a label at the beginning of a line. A label is a sequence of non-blank characters preceded on the line only by blanks and immediately followed by a blank, colon, or carriage return. For example, the following text lines have labels:

LI: FUNC()	label is LI
A B	label is A
END	label is END

Examples of label addressing are:

:AF /6/+:L2:	addresses first line that contains label L2 after the first line of block 6
:IT -:END:	addresses the first line before the current line that contains the label END

The last line of a block can be addressed. For example,

:LT /5/+$	addresses the last line of block 5
:ML $-1	addresses the line before the last line of the block that contains the current line

In general, a line address is a sequence of line address elements separated by plus or minus signs. Plus signs may be omitted where not syntactically ambiguous. An address element may be:

1) a block number enclosed in slashes

2) a function name for a block

3) a number which is a relative line displacement

4) a string enclosed in double quotes for context searches

5) a string enclosed in colons for label searches, or

6) a dollar sign for addressing the last line of the current block

Elements of types one and two are absolute while elements of types three through six address relative to the address determined by the elements to its left. The first element in an address is relative to the current line.

For the purpose of addressing, the first line of a block immediately follows the last line of the preceding block and the first line of the first block of a program immediately follows the last line of the last block of a program. For example,

:EL /1/-1	addresses the last line of the program
:IT /4/$+1	addresses the first line of block 5
:AT /7/-1	addresses the last line of block 6

3.2 Ranges

Some commands take a range as an argument. A range specifies a continuous sequence of text lines within a program. A range can be specified in four ways:

1) a pair of addresses separated by blanks. This specifies all lines between the first address and the second address, inclusive. The first address is determined relative to the current line and the second address is determined relative to the first address. The second address must not specify a line in the program before the line specified by the first address.
2) a single address. This specifies the single line addressed.
3) an "at" sign (@). This specifies all lines in the program.
4) no argument. This specifies the current line.

Examples of ranges are:

:LT /5/ /6/$	specifies all lines of block 5 and block 6
:DT Ø 2	specifies the current line and lines after the current line
:DT 1	specifies the line after the current line
:CB @	specifies the entire program
:DT	specifies the current line

4.0 Commands

There are over forty different commands that can be executed in the SIMPLE language system. The details of each command are given in the following sections.

In the form given for each command:

1) Upper-case characters are literal characters in the command line.
2) Lower-case names refer to constructs that are described in other parts of this document.
3) A construct inside square brackets means the construct is optional.
4) Constructs separated by vertical strokes indicate that any one of the constructs should be specified; and
5) (cr) is a carriage return and (lf) is a line feed.

4.1 Program Commands

Programs are specified by program-name. A program-name is a sequence of simple names separated by periods. A simple name is any sequence of characters not containing a period, asterisk, or carriage return (generally, just alphanumeric characters). A program-name uniquely specifies a search rule for finding the program and starting the search with the user directory for the logged-in user. If the program begins with a period, then the search begins with the system's root directory. If the program begins with an asterisk, then the search begins with the user's scratch directory. All segments in the scratch directory are destroyed when the user logs off the system. Assuming that JONES is the logged-in user, some examples of program-names are:

PROG1	find the segments of program PROG1 in directory JONES
DI.PROG	find directory DI in directory JONES and then find the segments of program PROG in directory DI
.SMITH.TEST	find the segments of program TEST in directory SMITH
*TEMP	find the segments of program TEMP in the scratch directory

The names of the text, debugger, and code segments of a program are formed by appending a number sign (#), slash (/), or "at" sign (@), respectively, to the end of the program-name.

Select-Program Command

form :SP program-name (cr) .

Causes the program with the specified name to be selected. All subsequent commands will act upon this program. A program must be selected before any command can be performed on it. A new program can be selected at any time.

New-Program Command

form :NP program-name (cr)

Causes the program with the specified name to be selected. If the program did not exist, then a text segment for the program is created. The text segment is initialized to a single line in block One containing END. Warning!! If the program already existed, then its previous contents are lost.

Destroy-Program Command

form :DP (cr)

Causes all the segments of the specified program to be destroyed. The specified program must be the same as the selected program or else the command will fail.

4.2 Text-Editing Commands

The text-editing commands provide the ability to construct a program by listing lines of text, adding new lines of text, deleting lines of text, modifying existing lines of text, moving lines of text from one part of a program to another or from another program, and performing string substitutions on the text.

List-Text Command

form :LT range (cr)

Causes all lines of text in the specified range to be listed on the terminal. The current line becomes the last line typed out.

Append-Text Command

form :AT address (cr)

Causes the subsequently typed lines to be appended after the specified address in the text segment. Upon typing the carriage return, the SIMPLE language system goes into "enter-text mode." Any number of lines of text can be typed in enter-text mode separated by carriage returns. Typing a line feed at the end of a line, instead of a carriage return, causes enter-text mode to be exited. The system will then prompt a colon and wait for the next command. A line feed typed immediately after a carriage return will not cause a blank line to be placed into the text. All line-collector features may be used during entry of a text line in enter-text mode. The old line is always the last line typed on the terminal. The current line becomes the last line appended.

For example,

```
:LT /2/ $ (cr)
ABC                          }
DEF                          } text before
GHI                          }
:AT /2/+1 (cr)                 append text command
123  cr
456  lf
:LT /2/ $
ABC                          }
DEF                          }
123                          } text after
456                          }
GHI                          }
:
```

Insert-Text Command

form :IT address (cr)

This command has an effect identical with that of the Append-Text (AT) command, except that the entered text is inserted before the line at the specified address and the current line becomes the line after the last inserted line.

Append-Function Command

form :AF address (cr)

This command has an effect identical with that of the Append-Text (AT) command, except that a new block is created after the block of the specified address and the entered text becomes the contents of this new block. The current line becomes the last line of the new block.

Delete-Text Command

form :DT range (cr)

Causes all lines in the specified range to be deleted from the text. The current line becomes the line that was before the deleted text. The range of lines must all be within the same block. If the range specifies all the lines in a block, then not only are all the lines deleted, but the block itself is also deleted; the current line becomes the last line of the previous block. It is not legal to delete all the lines of block One. For example,

```
:LT /3/ $ (cr)
LINE1                    }
LINE2                    }text before
LINE3                    }
:DT /3/+1 (cr)             delete-text command
:LT /3/ $ (cr)
LINE1                    }
LINE3                    }text after
:
```

Un-Delete Command

form :UD (cr)

If the immediately preceding command was a Delete-Text (DT) command then this command totally undoes all effects of that Delete-Text command. Otherwise, the command has no effect. This command provides protection against accidental deletions.

Modify-Line Command

form :ML address (cr)

The addressed line is modified under control of the line collector. The addressed line is the old line. The resulting new line replaces the previous contents of the addressed line. For example,

```
:LT /2/+2 (cr)
ABCD                    line before
:ML (cr)                modify line
XBCD (cr)               line is modified
:LT (cr)
XBCD                    line after
:
```

Edit-Line Command

form :EL address (cr)

Has exactly the same effect as the Modify-Line (ML) command except that the line to be modified is typed out just before the line modification begins. For example,

```
:LT /3/+3 (cr)
ABCD                    line before
:EL                     edit line
ABCD                    typed-out line
AXCD                    line is modified
:LT (cr)
AXCD                    line after
:
```

Copy-Text Command

form :CT address (cr)
SOURCE RANGE: range (cr)

Causes a copy of the lines of text in the specified range to be appended after the line at the specified address. The lines at the source range are not changed or deleted. The specified address must not be inside the specified range. The current line becomes the last line of the appended text.

Copy-Function Command

form :CF address (cr)
SOURCE RANGE: range (cr)

Has exactly the same effect as the Copy-Text (CT) command except that a new block is created after the block of the specified address and a copy of the text is placed in this block. The current line becomes the last line in the new block.

Transfer-Text Command

form :TT address (cr)
SOURCE PROGRAM: program-name (cr)
SOURCE RANGE: range (cr)

Has exactly the same effect of the Copy-Text (CT) command except that the text lines are copied from the specified source program. The specified source range is determined in the context of the source program. This command provides a way of copying text from one program to another.

Transfer-Function Command

form :TF address (cr)
SOURCE PROGRAM: program-name (cr)
SOURCE RANGE: range (cr)

Has exactly the same effect as the Copy-Function (CF) command except that the text lines are copied from the specified source program. The specified source range is determined in the context of the source program.

Replace-Functions Command

form :RF program-name (cr)

Provides a method for copying the text of entire blocks from the specified program to the currently selected program. Every text block in the specified source program is copied into the currently selected program. If either program is globally valid, then block One is not transferred. If both the source and destination programs are globally valid, and the function name for the source block matches the function name for some destination block, then the text of the source block replaces the text of that destination block. Otherwise, a new block is created at the end of the destination program and the source text is placed in this new block. The source program is not modified by this command. The current line does not change. A listing is typed out as each block is either "replaced" or "appended."

Substitute Command

form

:SU range (cr)
OPTION: [S|C|L] (cr)
SEARCH FOR: search-string (cr)
[REPLACE WITH: replace-string (cr)]

The specified range is searched for occurrences of the specified search-string. The search string and replace string may be any string of characters terminated by a carriage return. The option S, C, or L specifies what action is to be taken when each occurrence of the search string is found. For the S (substitute) option, the search string is replaced by the replace string and the search continues with no type out. For the C (confirm) option, the line address and the text line are typed out with carets delineating the occurrence of the search string within the line. The user now must confirm that he wishes the replacement by the replace string to occur by typing an S. Typing any other character will inhibit replacement. The search then continues. For the L (list) option, the address and text lines are listed for each occurrence of the search string but no substitution occurs. For all three options, the number of occurrences of the search string found is typed out at the end of the scan. If a string substitution would make the resulting line longer than 128 characters, then that substitution will not occur. The current line becomes the line where the last occurrence of the search string was found or is unchanged if there was no occurrence.

4.3 Compilation Commands

A program must be globally and locally valid before it can be executed. Initially, a program is globally invalid. A successful compilation makes it valid. Modifying the text of a block makes that block locally invalid. Creating or deleting a block makes the whole program globally invalid.

Compile Command

form :CP (cr)

Causes the program to be compiled. If the program is globally invalid then the entire program is compiled. If the global block (block One) or any function block headers had compilation errors then the program is left globally invalid and the individual function blocks will not be compiled. If globally valid, then only locally invalid function blocks are recompiled. A function block is left invalid if it had any compilation errors.

Set-Invalid Command

form :SI (cr)

Makes the selected program globally invalid.

Measurement-Mode Command

form :MM number (cr)

The measurement mode for the compiler is set to the value of the specified number. Initially, measurement mode is disabled. See Appendix D for measurement-mode details.

4.4 Program Execution Commands

The state of a program that is in execution or has been in execution is entirely stored in a pair of segments called the data segment and the capability segment. All program execution and debugging commands operate on the currently recovered state. A currently recovered state is established by the DO or RS commands described later.

A program may be in execution as either a primary computation or as a secondary computation. When a primary computation is evoked: (1) all attached terminals, open segments or created processes from previous computations are released, closed, and destroyed, respectively; (b) a new process is created for the user program to run in; (c) the program stack is reset; (d) the capability segment is cleared; and (e) control is transferred to the program by executing the specified immediate statement in the global context of the initialized state. If the primary computation stops because it trapped, encountered a breakpoint, or a panic was hit, then a secondary computation can be initialized. When a secondary computation is initiated, it does not effect the state of the primary computation; nothing is initialized. When a secondary computation stops, for whatever reason, its entire state is lost and the state of the stack before the secondary computation is recovered. However, the values of global variables may have been changed by the secondary computation.

An immediate statement may be any expression legal in the SIMPLE language. Unless the expression is a function call or an assignment, the result value of the expression is typed out on a normal completion of the computation initiated with the immediate statement. Upon abnormal

completions a stop message is typed out giving the address where the computation stopped and the reason for the stop.

Do Command

form :DO immediate-statement (cr)

Causes the program to begin execution as a primary computation. The specified immediate statement is executed in the global context (i.e., not in the context of any function). The immediate statement can be any expression legal in the SIMPLE language. Normally, this will be a function call on the driver function of a program. For example,

```
:DO F() (cr)          start primary computation
F+17:  BREAK          stop message
:
```

Immediate Statements

form : immediate-statement (cr)

Normally, an immediate statement preceded by a blank is executed as a secondary computation. However, if no primary computation exists, then it is executed as a primary computation. The secondary immediate statement is executed in the context of the current function activation. Normally, this is the function executing when the primary computation stops, but it can be changed using the IN command. If a secondary immediate statement executes a GOTO, RETURN, or FRETURN operation, then control is returned to the primary computation.

Go Command

form :GO [number] (cr)

If the primary computation has stopped at a breakpoint, then execution of the primary computation is resumed by the GO command. The number, if present, (assumed one if omitted) indicates the number of breakpoints that must be encountered during execution before the computation will again stop at a breakpoint.

4.5 Breakpoint Commands

Breakpoints may be set on any line in a program that begins with an executable statement. If a breakpoint is set on a line, then execution of the computation is stopped just before that line would be executed.

Set-Breakpoint Command

form :SB address (cr)

Causes a breakpoint to be set at the specified address. The current line becomes the addressed line.

Clear-Breakpoint Command

form :CB range (cr)

Causes all breakpoints in the specified range to be cleared. The current line becomes the last line of the specified range.

List-Breakpoints Command

form :LB range (cr)

Causes the addresses of all breakpoints set within the specified range to be printed out. The current line becomes the last line in the specified range.

4.6 State Commands

A current state can be established with an RS command or by starting a primary computation.

Whenever a primary computation is initiated, the current state is established as a data segment and a capability segment in the scratch directory with names derived by appending a dollar sign or a colon, respectively, to the rightmost simple name of the currently selected program name. For example,

program name	state segment names
PROG	*PROG$ *PROG:
JONES.LIST.TEST	*TEST$ *TEST:

Recover-State Command

form :RS [name] (cr)

Causes the current state to be the two segments with names formed by appending a dollar sign or colon, respectively, to the specified name. If the specified name is omitted, then the rightmost simple name of the currently selected program name is assumed.

Save-State Command

form :SS name (cr)

Causes the current state segments to be copied to two segments with names formed by appending a dollar sign and a colon to the specified name.

4.7 Debugging Commands

Display-Stack Command

form :DS [P] (cr)

Causes the nested state of function activations in the stack of the primary computation to be typed out. Each line is the address from which the next function was called. The top line is the address where the computation is stopped. If the P option is specified, then the function-base relative P-counter values of the calls are also typed out in octal. For example,

```
:DS (cr)
F1+12                   address where program stopped
GFUNC+2                 address of call on F1
DRIVER+20               address of call on GFUNC
:
```

In Command

form :IN function name [number-n] (cr)

Whenever the primary computation stops execution, the current function activation becomes the top activation on the stack. The IN command allows selection of a different activation on the stack as the current activation. The new current activation is determined by scanning from the top of the stack for the n-th occurrence of an activation for the specified function name. If number-n is omitted, then one is assumed.

Print-Value Command

form :PV value-name [D|B]

Causes the named value to be printed. If the value-name is defined for the current function activation, then that value is printed. Otherwise, if the value name is globally defined then the global value is printed. The value is printed in octal as two halfwords (4 halfwords if value is of type descriptor) unless a D or B option is specified. The D option causes value to be printed in decimal as a 32-bit word (2 words if descriptor). The B option causes the value to be printed in octal as four bytes (8 bytes if descriptor).

Print-Segment Command

form :PS [segment-name] [lower-bound] [upper-bound] [D|B] (cr)

Causes the values of the words of the specified segment in the range specified by the lower-bound and upper-bound to be printed. The lower-bound and upper-bound must be octal addresses. If the bounds are out of the segment range they are adjusted to be in range. The upper-bound should not be less than the lower-bound. If the upper bound is omitted, then it is set to the lower-bound. If the lower-bound is omitted, then it is assumed to be zero. If the segment-name is omitted, then the segment specified in the previous PS command is assumed. A D, B, or null final option causes words to be printed in decimal, as four octal bytes, or as two octal halfwords, respectively.

Print Data-Segment Command

form :PD [lower-bound] [upper-bound] [D|B] (cr)

Has exactly the same effect as the Print-Segment (PS) command except that the values are printed from the data segment of the currently recovered state.

Fill-Segment Command

form :FS segment-name address value-list (cr)

Causes the values specified in the value-list to be stored in successive words of the specified segment starting at the specified address. A value-list contains an arbitrary number of values separated by blanks. A value is specified as two octal numbers separated by a colon. The left number goes in the left half of the word and the right number in the right half. The address and new value of each modified word is printed.

4.8 Miscellaneous Commands

Exit Command

form :EX (cr)

Causes SIMPLE language system to be exited.

Where Command

form :WH (cr)

Prints the address of the current line.

Commands-from-Segment Command

form :CS segment-name (cr)

Causes the SIMPLE language system to accept subsequent commands from the specified segment. The segment should be a text segment. The successive commands are taken from successive lines of block One of the segment. The SIMPLE language system will resume taking commands from the terminal when all commands in block One have been executed or after an attention panic is hit.

Symbol-Table Command

form :ST name (cr)

Causes the contents of the symbol-table entry for the specified name to be printed. First, the symbol table for the current function activation is searched for the name, and then the global symbol table.

Print-Code Command

form :PC address (cr)

Causes the line-table entry and compiled code (as octal halfwords) for the specified line to be printed. The current line becomes the addressed line.

Select-Runtime Command

form :SR runtime-name (cr)

Causes the program with the specified runtime-name to be selected as the current runtime. Initially, the standard runtime is selected. Whenever a globally-valid program is selected with the SP command, the runtime with which it was globally compiled is selected. The SR command provides a way of overriding this selection. A program is globally compiled using the currently selected runtime.

Runtime-Mode Command

form :RM (cr)

Places the SIMPLE language system in runtime mode. Runtime mode causes the selected program to be compiled as a runtime, and allows addressing of programs compiled as runtimes by function name as well as by block number.

User-Mode Command

form :UM (cr)

Causes the SIMPLE langauage system to return to user mode (its initial state) from runtime mode.

5.0 Attention Panics

If a computation is not in execution, then hitting the escape key on the keyboard will cause an attention panic. An attention panic causes immediate termination of any command in progress. This means that commands taking a long time to perform their action may be prematurely terminated. In all cases, a message is printed indicating what termination action was taken. If an attention panic occurs:

1) During entry of a command line, then the message PANIC! is printed. The partial command is ignored and the system prompts for the next command.

2) While in enter-text mode (during IT, AT, and AF commands), then the message PANIC! is printed. All completed lines are entered into the text, the last partial line is ignored, and the system prompts for the next command.

3) While waiting for input in commands that expect input (i.e., ML, EL, SU, CT, TT, TF, amd DP commands) no further action is taken in the command. The message COMMAND ABORTED! is printed and the system prompts for the next command.

4) During commands that generate lists (i.e., LT, LB, DS, PS, SU, PD, PC, and RF commands), the listing is terminated immediately. The message PANIC! is printed and the system prompts for the next command. The listing in the RF command indicates exactly which functions were copied.

5) During a compilation, the compilation is immediately terminated. The message COMPILATION ABORTED is printed and the system prompts for the next command. The compilation can be restarted with another CP command.

6) While a computation is in execution, then the attention panic can be fielded by the running program and therefore, the action taken is determined by that program. If the running program is not fielding its own attention panics, then the computation stops and a stop message is printed.

7) While in commands from segment mode, then the system returns to commands from terminal mode at the end of the current command.

8) In all other cases, the command in progress completes its action, the message PANIC! is printed, and the system prompts for the next command.

6.0 Debugger Panics

A computation in execution may be immediately stopped by typing a control right bracket ($]^c$) on the keyboard, thus causing a debugger panic. A stop message giving the address of the current line of execution in computation is printed. If there is not a computation in execution, then a debugger panic on the terminal is ignored except that the terminal bell will sound.

APPENDIX A - Command Summary

Program Commands

SP	program-name	select program
NP	program-name	new program
DP		destroy program

Text-Editing Commands

LT	range	list text
AT	address	append text
IT	address	insert text
AF	address	append function
DT	range	delete text
UD		un-delete
ML	address	modify line
EL	address	edit line
CT	address	copy text
CF	address	copy function
TT	address	transfer text
TF	address	transfer function
RF	program-name	replace functions
SU	range	substitute

Compilation Commands

CP		compile program
SI		set invalid
MM	number	measurement mode

Program-Execution Commands

DO	immediate-statement	execute primary computation
sp	immediate-statement	execute secondary computation
GO	[number]	continue primary computation

Breakpoint Commands

SB	address	set breakpoint
CB	range	clear breakpoint
LB	range	list breakpoints

State Commands

RS	[name]	recover state
SS	name	save state

Debugger Commands

DS	[P]	display stack
IN	function-name [number]	select current function activation
PV	value-name [D|B]	print value
PS	[segment-name] [lower-bound] [upper-bound] [D|B]	print segment
PD	[lower-bound] [upper-bound] [D|B]	print data segment
FS	segment-name address value-list	fill segment

Miscellaneous Commands

EX		exit language system
WH		where is current line
CS	segment-name	commands from segment
ST	name	symbol table print
PC	address	print code
SR	runtime-name	select runtime
RM		runtime mode
UM		user mode

APPENDIX B - Runtime Debugging Functions

The following functions included in the standard runtime can be used to print the values in common data structures of SIMPLE programs when executed as immediate statements. Two predeclared global variables, \M and \R, determine the form of value printout.

\M = 'F'	full word
\M = 'H'	two half words (initial value)
\M = 'B'	four bytes
\M = 'C'	four characters

\R = radix for printout in \M = F, H, or B cases (initially 8)

Print-String Function

form PS(string)

Causes the contents of the specified string to be printed. Byte strings are always printed as a string of characters.

Print-Vector Function

form PV(vector [,lower-bound][,upper-bound][,mode])

Causes the specified vector to be printed within the specified range of indices. The mode, if specified, will override the state of \M. If no bounds are specified, then the entire vector is printed. If only a lower-bound is specified, then only that single value is printed.

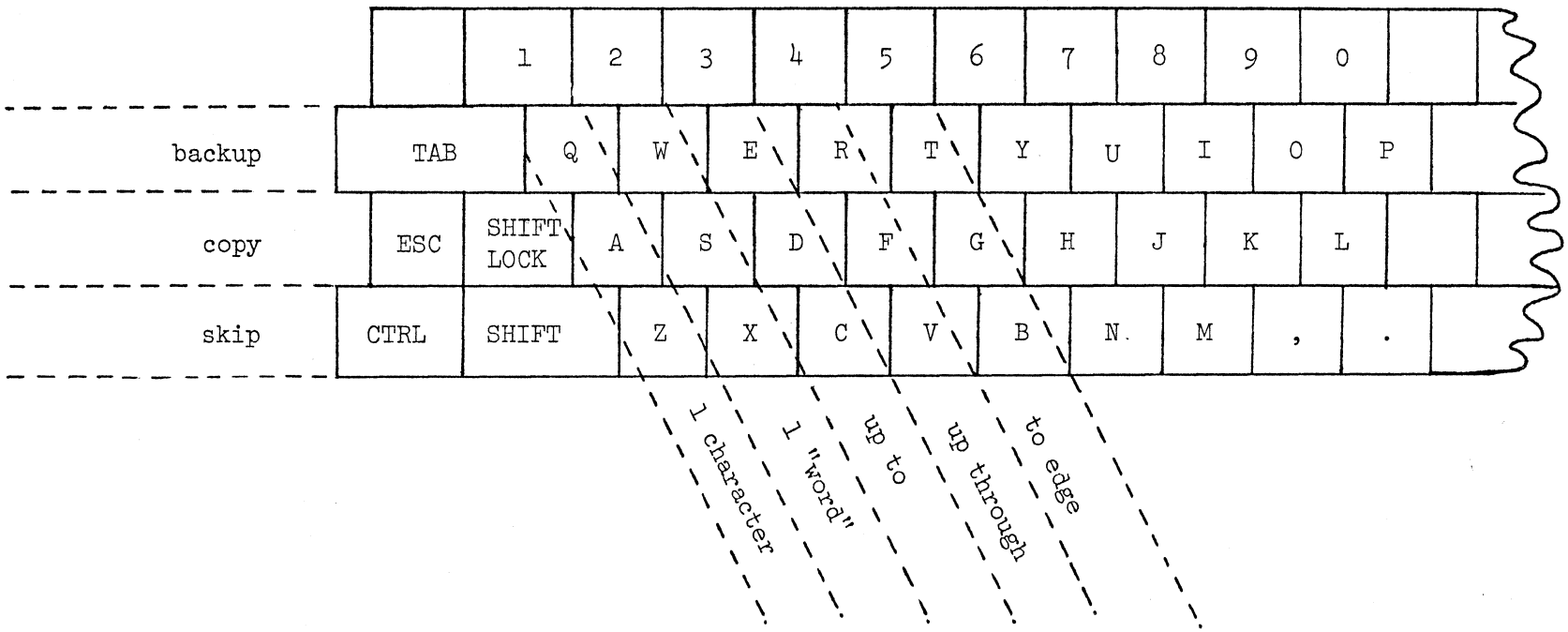
Print-Reference Function

form PR(reference)

Causes the values pointed to by the specified reference to be printed.

Note: The \M and \R variables are used to determine the form of printout for the value printed on normal termination of immediate statement execution when the immediate statement is not a function call or an assignment.

CRMS LINE COLLECTOR REQUESTS



CARRIAGE RETURN	accept new line as input
LINE FEED	accept new line as input, with special termination flag
TAB	advance to next tab stop

Y^c	toggle insert mode
U^c	toggle underline mode
I^c	(same as TAB)
O^c	interpret next two characters as an overstrike
P^c	print remaining old line, concatenate it to new line, and accept new line

H^c	(same as backspace) no effect
J^c	(same as line feed)
K^c	concatenate remaining old line to new line and accept new line
L^c	accept next character without line collector interpretation
N^c	restart input, using new line as the old line
M^c	(same as carriage return)

APPENDIX D - SIMPLE Program Measurement

The SIMPLE program measurement system allows evaluation of the time spent in the various functions of a program. In order to use the measurement facility, the program must first be recompiled globally in measurement mode.

Example:

```
:SP PROG
:MM 1  (or MM -1)
:SI
:CP
```

Any number of the user functions of a program may be specified to be measured. This may be done by appending MEASURE to the end of the FUNCTION statement for a function to be measured and then recompiling the function. Alternatively, a user may specify that all user functions be measured by recompiling the program globally after doing an MM -1 command.

Example:

```
FUNCTION F(X,Y,) RETURNING VECTORS MEASURE
```

Three values are maintained about each measured function:

1) The accumulated elapsed real time spent in the function in milliseconds. This time is measured from the time the function is called until it returns or until some other measured function (perhaps a recursive call on itself) is called. When this function returns, real time is once again accumulated. That is, time is accumulated for the topmost measured function on the call stack.

2) the CPU compute time spent in the function in milliseconds. It is measured in the same way as real time.

3) the number of times the measured function is called.

To begin measuring a program, the program must execute a call on runtime function INITMEAS. It takes a single Boolean-valued argument (default = TRUE) which specifies if CPU elapsed-time measurements are to be taken. Typically, INITMEAS will be called from the beginning of the program's driver function.

Example:

```
INITMEAS(TRUE)
```

The accumulated statistics may be printed out by calling runtime function PRINTMEAS as an immediate statement while the state of the measured computation is recovered.

Example:

```
:SP PROG
:RS
:DO PRINTMEAS()
```

APPENDIX E - Stop Messages

The following messages may be printed following the stop address when a computation stops. The normal meaning of the message is given.

ILLEGAL FAIL RETURN

attempt to fail return from function when not permitted in calling statement

ILLEGAL VECTOR OPERATION

attempt to use an out-of-bounds index or use an invalid vector descriptor

ILLEGAL FIELD OPERATION

attempt to use an out-of-bounds index or use an illegal descriptor

ILLEGAL FUNCTION CALL

attempt to make an illegal function call

ILLEGAL RING OPERATION

attempt to access a ring with an illegal vector descriptor or ring selector

BAD OPCODE TRAP: n

attempt to execute an illegal instruction with octal opcode value n (frequently an attempt to store into a constant)

BREAK

the program has encountered a breakpoint

STATIC BREAK

the program executed a BREAK statement or encountered an inconsistent breakpoint

ARITHMETIC TRAP

an arithmetic overflow occurred during an add, subtract, multiply, or divide

PANIC

an unfielded attention panic occurred

DEBUGGER PANIC

a debugger panic occurred

DONE

the program terminated with a call on the HALT function

WINDOW TRAP

an attempt to access a word out-of-bounds of a segment occurred

STACK-OVERFLOW TRAP

the size of the program stack was exceeded

SUBPROCESS TRAP #n

a trap of type n (octal) occurred in a subprocess created by the debugged program

START-UP IMPOSSIBLE

ATTENTION TRAP

P-COUNTER TRAP

QUANTUM-OVERFLOW TRAP

INSTRUCTION-TRACE TRAP

FUNCTION-TRACE TRAP

all these messages probably indicate a system error has occurred