Name: Mark Brown

Project: 1 Programmer: MRB

File Name: RAID.PMP[S,DOC]

File Last Written: 13:19 28 Apr 1976

Time: 0:03 Date: 15 Aug 1976

Stanford University
Artificial Intelligence Laboratory
Computer Science Department
Stanford, California

```
COMMENT ⊗   VALID 00020 PAGES
C REC  PAGE   DESCRIPTION
C00001 00001
C00003 00002    INTRODUCTION, LOADING RAID
C00010 00003    THE RAID DISPLAY, ERROR MESSAGES, COMMAND FORMAT
C00016 00004    VALUES sp  λ + - * / () , ,, @ α. β. = " α" ' α' α% &
C00025 00005    SCREEN CONTROL αI βI ϵI cr αcr βcr ϵcr
C00027 00006    DISPLAY MODES C O D F H T λT Q V λV A U λU λJ λR λL
C00038 00007    SYMBOL TABLE COMMANDS λα: λα& λαZ βZ λ: λ←λ λαK λϵ: λβK
C00044 00008    OPENING CELLS  λsm λαsm αsm βsm > < LF \ α< α> β< β> ϵ< ϵ> α← λα= λ≡
C00051 00009    DEPOSITING IN CELLS λcr λ> λ< λα> λα< λβ< λβ> λϵ< λϵ> βE λβE
C00058 00010    OPENING INDIRECT LOCATIONS tb λtb sm α[ α] α@ β[ β] β@
C00062 00011    DYNAMIC LOCATIONS ϵ[ ϵ] ϵ@ λϵ[ λϵ] λϵ@
C00074 00012    SEARCHES λαW λαN λαE ∨ ∧ λU λαU λn λαn
C00085 00013    PROGRAM CONTROL λαG αG λαB λβB αP λαP βP
C00093 00014    STEPPING, EXECUTING αS βS αX βX ϵX ϵY ϵE
C00103 00015    MULTI-STEP  ϵS λϵS X S
C00107 00016    MACROS  λαM∂ λβM ϵ∂
C00109 00017    SHORT SUMMARY OF RAID COMMANDS
C00115 00018    APPENDIX: SYMBOLIC MODE, PRINTING SYMBOLS, BYTE SIZE 0
C00121 00019    APPENDIX: RAID DEFINED LOCATIONS AND TABLE OF POINTERS
C00128 00020    APPENDIX: BREAKPOINTS
C00139 ENDMK
C⊗;
```

INTRODUCTION, LOADING RAID

This document was written by Phil Petit and revised in March, 1975 by Raphael Finkel.

INTRODUCTION

RAID is an interactive debugging program that uses the displays and allows dynamic monitoring of memory locations. RAID lives in core with your program and allows you to do various things with and to your program, such as stop at selected places and examine your core image, etc. The major advantage of RAID over DDT is that RAID uses the displays to give you a constantly updated view of selected locations in core. It also can be used from a teletype, typing only those lines that have changed since last time it wanted to display something.

RAID was written by Phil Petit. It was modified by Dan Swinehart to use a sorted symbol table. Jeff Rubin has added further inprovements and fixed some bugs.

LOADING RAID

It is possible to use RAID in a stand-alone fashion, for writing small programs on the spot, or finding the octal representation of something, for example. The monitor command R RAID calls in a fresh copy of RAID and starts it up.

RPG knows about RAID; the PREPARE, TRY, and DEBUG commands cause RAID to be loaded with your program. This saves you the trouble of remembering the loader switches.

The loader will load RAID with your program if you use the /V or /H switch. This saves you the trouble of explicitly loading RAID.REL[1,3]. For example if the binary file for your program was called FNORP, the loader command /VFNORP$ (where $ means alt-mode) would cause the loader to load RAID with your program. If you put this switch after the name of your file, it is also necessary to tell the loader to load your program symbols; this is done with the /S switch. Here are some sample commands to load FNORP:

```
/SFNORP/V$        (Load symbol table, then RAID)
/VFNORP$          (Load RAID, then FNORP, then symbol table)
```

It is sometimes a good idea to put RAID after all the code, so addresses in the core image will be close to where they would be without the RAID. This helps if you are debugging a copy without RAID in parallel to one with RAID.

Once you have your program and RAID in core, you must get into RAID. This is done by typing DDT<cr> to the system. This is because the system can't tell the difference between RAID and DDT. It is also possible to enter RAID by hitting a breakpoint (ways to set and clear breakpoints are discussed later) or by jumping to RAID's starting address (this is the global symbol DDT, and it can also be found in the right half of JOBDDT).

When RAID is entered, the display screen flashes and the RAID display will appear. The duplexing of keyboard input moves down to the bottom of the screen. RAID is now ready to accept commands.

There is also a version of RAID for examining files; it is called

FRAID. The monitor command R FRAID calls in a fresh copy and starts it up. FRAID first asks if the file to be examined is a DMP file (if so, the starting address will be noted and FRAID will look for a symbol table) and whether the file is to be modified. If FRAID is run in file-modify mode, all changes made in the core version of the file are immediately written onto the disk.

All of the commands that run a job and set and clear breakpoints are illegal in FRAID. Furthermore, if the file is a dump file, locations 0 through 73 are non-existant. There are a few commands that work only in FRAID or which have different meanings in FRAID. These are:

αS Simulates setting $IO to -1. Causes I/O instructions to be typed out

βS Simulates setting $IO to 0. Causes I/O instructions to not be typed out

εE Exits from FRAID. RELEASES the file after finishing all modifications.

THE RAID DISPLAY, ERROR MESSAGES, COMMAND FORMAT

RAID maintains a display of two columns. The left column gives names of addresses currently being examined; the right column displays the contents of those addresses. Commands are available for determining what locations to display, how many to display, and in what mode to display them.

Keyboard echoing is moved down to the bottom of the screen. RAID uses special activation mode; this means that some characters like ";" and "<" activate. It can lead to very unfortunate results if you activate your commands with a carriage return, since that has a special (and sometimes irreversable) meaning.

At the top of the screen is either a large OK, a large *, or an error message. RAID complains if it does not understand your command by presenting a large "?" in this location on the screen. If you refer to a symbol FOO and it is not known to RAID, the complaint will look like "U -- FOO", which means FOO is undefined. If RAID knows about several symbols FOO, then the error is "M -- FOO" meaning FOO is multiply defined. RAID will use one of the meanings anyway: the most deeply nested one it can find in the current block. Other messages, like "SKIP 2" can appear here, too. These will be discussed where appropriate.

At the head of the left column is the name of the program whose symbol table is currently being used. If you load several modules together, RAID will only use the symbols from one at a time; there are commands to switch to another one. If your program has block structure, then RAID can be directed to a particular block in your program (to disambiguate symbols with the same name). The current block name is shown at the top of the right column.

Addresses in the left column that are not in the current program are given in octal; those in the current program are given as offsets from labeled locations (if there is one of those close enough), or in octal as a last resort. Symbols not in the current block are presented with the block name appended, so if FOO is in block BAZ and we are now in block GARP, the location after FOO would be displayed as BAZ&FOO+1, assuming that BAZ does not contain GARP.

Lines are sometimes marked with special signs in the space between the columns. These signs are "." (dot), "→" (arrow), "x" (ex), and "↔" (swap). Swap means that dot, arrow, and ex all appear on the same line. RAID on a teletype uses "<" for ex, ">" for arrow, and "<=>" for swap.

Dot points to the line that is currently "open"; many commands refer to the dotted line. The significance of being open is that it is possible to deposit new information in the open word. Arrow points to the line that is currently "being looked at", but it cannot be modified unless the dot is moved there. Ex points to the line holding the next instruction to be executed if it is on the screen. This can happen if RAID is invoked while the program is in execution or when a breakpoint is encountered.

Many RAID commands involve control bits. For the sake of clarity and conciseness, we adopt the following conventions for this document: The symbol α means <control>, β means <meta>, and ϵ means <control-meta>. These can be simulated by **$**, **$$**, and **$$$**, respectively, where **$** means <altmode>. In addition, λ will be used to refer to a value. Thus $\lambda\beta[$ means some value (without any control

bits), followed by <meta>[. We will also have occasion to use the sign ∂ to refer to any character.

VALUES sp ⊗ λ + - * / () , ,, @ α. β. = " α" ' α' α% &

Commands to RAID often have parts that consist of values. By convention, we will refer to values by the symbol λ in this discussion. A value can be:

A string of octal digits
    Examples: 463, 400321

A string of decimal digits preceeded with =
    Examples: =293, -=30098
    Note that it is not legal to say =-30098.

Two strings of decimal digits separated by a dot. (i.e. a floating point number.
    Examples: 123.456 0.003 17.
    Note that .123 is not a valid floating point number.

A defined symbol in your program
    Examples: FOO, BAR

⊗
: If the character ⊗ is typed then the following characters are used as an identifier even if they wouldn't have been had they been typed without the ⊗. This is useful for identifiers starting with digits. An example is ⊗13P, which is the identifier "13P".

&
: If you have several labels with the same name, but in different blocks, you can specify which block you mean with this symbol. For example, the variable FOO in block BL1 can be called BL1&FOO. There are also commands for setting the default block; see the page on symbol table manipulation.

$M $C $nB etc
: RAID has some predefined symbols that are of use to the sophisticated user. See the page on RAID defined symbols.

A machine instruction
    Examples: MOVE, HRLZI A,FOO+3

A UUO
    Examples: BEEP B, SPWBUT A,

A string constant, in one of several modes:

"
: " followed by a chr., followed by a string not containing that chr., followed by that chr., has the value of the left adjusted ascii of the first 5 or fewer characters in the string. For example, "/POT/ has the same value as ASCII /POT/ has in the assembler, namely -- 502372400000

α"
: is just like " except that up to =17 words can be typed between the delimiters. (Extra characters are discarded.) These words are deposited in successive locations (starting with dot) when you type a carriage

return (this is one of the commands used for modifying cells). If carriage return is not used to terminate the command, the whole string is discarded.

'

is just like " except that it is sixbit. For example, '/DSK/ has the same value as SIXBIT /DSK/ has in the assembler, namely -- 446353000000

α'

this causes the string of characters following it, up to the first non-letter non-digit character to be converted to radix50, and has that value. Only 6 characters are used.

A byte constant:

α%

followed by a string of values (not containing comma) separated by commas, causes the values in the string after the first to be considered bytes of a size indicated by the first value in the string. For example, α%3,5,4,3,7,0,7,1 has the value 543707100000. If the byte size is zero, the byte descripter is taken from a special location in RAID. This is discussed in the appendix.

A byte pointer constant:

β.

followed by two or three values (not containing comma) separated by commas, causes a byte pointer to be assembled with the same value that FAIL would generate for POINT VAL1,VAL2,VAL3 with the exception that decimal radix is not forced for VAL1 and VAL3 as in FAIL. If the trailing comma and VAL3 are omitted, then 44 octal is used for the position field.

A special symbol:

Here is a list of the special symbols:

@ @ has the value 20,, i.e. the indirect bit

. "." has the same special meaning as in the assembler i.e. the place you currently are. This line is marked on the display with the dot.

α. has the value of the contents of dot.

λα. has the value of the contents of λ .

An arithmetic expression involving any of the above:

Here is a list of the legal arithmetic expressions. Note that all arithmetic is integer:

λspλ that is, the first value, a space, and then the second one. The value of this is the sum of the two values except that if the second one is an octal constant, it is truncated to =18 bits.

λ+λ sum of the two values, =36 bits

λ-λ	difference of the two values
λ*λ	product (multiplication)
λ/λ	quotient (integer division)
(λ)	this causes the two halves of λ to be swapped and all =36 bits added to the rest of the expression.
λ,	if not followed immediately by another comma, this causes the value to be truncated to four bits and shifted left =23 bits into the accumulator field.
λ,,	this causes λ to be placed in the left half (i.e. shifted left 18 places).
,λ	this truncates λ to =18 bits.
,,λ	this truncates λ to =18 bits.

SCREEN CONTROL αI βI ϵI cr αcr βcr ϵcr

When Raid is run on a Data Disc terminal it notices whether you have cleared the screen recently or not (esc C, break P, break N). If you have, then Raid will output the entire display again the next time it executes a command. There are some commands that explicitly change the display:

λαI
This causes the number of lines RAID is willing to display to be set to λ. This number is initially =21.

αI
This causes the number of lines RAID is willing to display to be increased by 1. The limit is =21. RAID starts with the full complement.

βI
This causes the screen to be cleared of all displayed locations. It does not change the number of locations RAID is willing to display.

ϵI
This is the same as βI except that protected locations are not cleared.

cr αcr βcr ϵcr
Note that all forms of carriage return are equivalent.

Carriage return, not preceded by a value, causes the screen to be updated and refreshed.

DISPLAY MODES C O D F H T λT Q V λV A U λU λJ λR λL

Locations are opened and displayed in various formats, called modes. RAID initially opens locations in symbolic mode This means that words are displayed, as much as possible, as machine instructions in standard assembler format. All this can be changed by the mode commands listed below. All the mode commands use one or more control bits (they are all typed with one or more control keys), and the significance of the different combinations of control bits is the same for all of them. If a mode command is typed with only the α bit, the mode of the location pointed to by the arrow is changed, and nothing else. If it is typed with β, then the mode in which future locations (that is, locations not yet displayed anywhere on the screen) are opened is changed, until a carriage return is typed, at which point the mode reverts to the "standard" mode -- initially symbolic. If it is typed with both control bits (ϵ) the mode for future locations is changed as with β, but the "standard" mode is also changed. Note that with both the β and the ϵ, none of the modes of locations already on the screen is changed.

C

This sets the mode to symbolic. This means that words will be displayed as machine instructions, if possible, and that fields (address, index, etc.) will be displayed as symbols from your symbol table (see below) plus or minus an offset, if possible. See the page on symbolic mode for a full discussion of the parameters that determine what it does.

O

This sets the mode to octal. Words are displayed as octal (base =8) numbers, except that a space is inserted between the left and right halves of words if the left half is not zero. Numbers are always displayed as positive quantities, so -1 is shown as "777777 777777".

D

This displays words as decimal numbers, preceded by an equal-sign (=). Negative quantities are shown properly; -1 is displayed as "-=1".

F

This displays words as decimal floating-point numbers, unless they are not normalized, in which case they are displayed in decimal mode (above).

H

This is half-word mode. The left and right halves of the word are displayed, symbolically (i.e. as symbols from your symbol table, plus or minus an offset), separated by a double comma, except if the left half is zero, in which case only the right half is displayed.

T

This is ascii, or 7-bit type-out mode. The word is considered to contain 5 characters of 7 bits each, left adjusted in the word. These 5 characters, plus a possible sixth ("@") if the low order bit is on, are displayed. If the high order 7 bits (first character) in the word is 0 (null), the word is considered to contain right adjusted ascii. Note that otherwise, nulls are ignored; the word 404010200206 will be displayed as "ABC ", not as "A B C".

Carriage return, line feed, tab, and backspace appear on a datadisk as a small CR, LF, TB, and BS, respectively, on a III as a small CR, LF, TAB, BS, and on a teletype as <<CR>>, <<LF>>, <<TAB>>, <<BS>>. An altmode is displayed as <<ALT>> on a teletype.

λT

This is for other character type-out modes. Legal values for λ are: 7 for ascii (as above), 6 for sixbit, and 5 for radix50. The extra 4 bits on the left of a word in radix50 are shown as two octal digits to the left of the string.

Q

This is byte-pointer mode. The word is displayed in exactly the format used by the assembler POINT pseudo-op, that is, the word POINT followed by a size field, followed by a comma, then the address, index, etc., then possibly another comma and a position. The size and position are decimal.

λV

This is byte mode -- the output analog of the α% input mode. The word is typed out as an appropriate number of bytes, separated by commas. The bytes themselves are typed in octal. The λ is the byte size and should not be negative. A byte size of 0 has a special meaning that is described in the appendix.

A

This is absolute mode. This one is different from all the others in that it does not change the basic mode of the of the word displayed, but merely causes addresses and other fields to be typed as numbers instead of symbols. One way to get out of this mode is to switch to C mode.

U

This is the same as symbolic mode except that the address field (right half of the word) is considered to be up to three right-adjusted ascii characters and is typed that way.

λU

This is as above except that λ indicates: if it is 5, then radix50 for the address field, if it is 6, then sixbit, and if it is 7, then ascii, as above.

λJ

This is flag mode. If certain bits are being used with special meanings in a flag word, it is nice to know exactly which flags are on. Each of the =36 bits in the word can be given a special name. The table of names should be in radix50 and occupy =36 words; the first of which refers to bit zero, the high-order bit. A zero entry in this table means that the bit has no special meaning.

In order to point RAID to the name table, a table pointer should be placed in location $M+3 in RAID. The right half should point to the table. The left half can point to another table pointer in the same format as the one in $M+3; thus you can link together many flag tables. Which table is used depends on λ. If λ is absent, the first table will be used. (This is table zero.) Otherwise, the linked list whose header is in $M+3 will be traced to find the specified table. If the list stops before then (the left half of some table pointer is zero) then symbolic mode is used.

Words displayed in flag mode have the names of all the bits that are on listed, separated by ! (which means "or" to the assembler and is not a legal radix50 character), followed by the octal corresponding to unnamed bits. If you change the table, RAID will not update those words being diplayed in flag mode, but newly displayed locations will be treated properly.

λR

This is right flag mode. It is just like J, except only the right half of the word is treated as flags; the left half is displayed in symbolic.

λL

This is left flag mode. It is just like R, except that the right half of the word is treated as left flags.

SYMBOL TABLE COMMANDS λα: λα& λαZ βZ λ: λ←λ λαK λε: λβK

λα:

This points RAID at the symbol table for the program indicated by λ. In this case, λ should be a single identifier. The name of the program will be shown at the top of the left column.

λα&

If you don't use block structure, ignore this command. It points RAID at the symbols for the block indicated by λ, which again should be a single identifier. If the block λ is defined in the current program, that one will be used; otherwise, the correct program will be chosen and its name dispalyed at the top of the left column. The name of the block will be shown at the top of the right column. All symbols will be displayed appropriately for this block.

λαZ

A list is kept of the names of the last few blocks opened. This command goes back λ blocks, and re-opens that one. There is never more than one copy of the same block name in the list. If λ is omitted, 1 is assumed.

βZ

The above list is forgotten. Then a new one is started, beginning with the currently open block.

λ:

λ should be a single identifier. This identifier is created and placed in your symbol table in the block you are currently inside. The address of the identifier will be the current value of dot, i.e. the location currently pointed to. Since ":" does not activate, this command will not take effect until the next activation character, so it is good practice to end this command with a carriage return. Note that if λ is already in the symbol table, then this command redefines it.

λ1←λ2

This command differs in format from the standard. λ1 should be a single identifier; λ2 may be any value. This command creates a symbol with the name λ1 and sets its address to λ2. λ2 should be followed by a carriage return. Note that if λ1 is already in the symbol table, then this command redefines it.

λαK

This causes the symbol λ (which should be a single identifier) to have a bit turned on in its symbol table entry that causes RAID to not use the symbol for typing out the contents of locations. This is called half-killing the symbol. In other words, this has the same effect as using double left arrow (←←) in the assembler.

λε:

This removes the symbol table bit that half-killed the symbol, thus resurrecting it. It will now be used for typing out contents of locations.

λβK

This causes the symbol λ to be deleted from the symbol table.

OPENING CELLS λsm λαsm αsm βsm > < LF \ α< α> β< β> ϵ< ϵ> α← λα= λ≡

λ;
This causes the location λ to be opened. If it is already on the screen, RAID just moves dot and arrow to that position, otherwise, it displays it in the next location on the screen. Both dot and arrow are moved to the opened location.

λα;
This causes the location λ to be opened as above, except that the location is also protected. A star appears on the screen to the left of the protected location; that location cannot be erased from the screen. If you protect too many locations, then RAID will refuse to open any new locations until there is room on the screen.

α;
This command causes the location RAID is currently pointing to with the arrow to be protected.

β;
The location RAID is currently pointing to with the arrow becomes unprotected. Any argument is ignored. You must be pointing to the location you want unprotected.

> LF
< \
Note that linefeed is always equivalent to >, and \ is always equivalent to <. This is true for all combinations of control keys.

These commands cause RAID to open the next higher (>) or lower (<) location from the one with the arrow. Both the dot and the arrow are moved to this location. For example, if it is currently pointing at location 36, > would cause it to display (and point at) location 37. If the location is not already on the screen, the pointer moves down (>) or up (<) one position to display this next location, wrapping around the screen if necessary.

α> αlf
α< α\
These cause RAID to move its dot and arrow down (α>) or up (α<) one position from the current location of arrow without changing the display in any other way. As always, the pointer and dot wrap around the screen if necessary.

β> βlf
β< β\
These are like > or < without any control, except that the new location is displayed in the same mode as the location marked with the arrow. For example, if you open the first word of your teletype buffer, and change the mode for it to ascii, you can then open the second word of the buffer in ascii by using this command.

ϵ> ϵlf
ϵ< ϵ\
These are the same as α> and α< except the location to which the arrow and dot are moved is displayed according to the mode of the line that currently has the arrow.

α←

This causes RAID to open, on the screen, the next location your program is going to execute, i.e. the next step location, or the last break-point you hit. The arrow, the dot, and the ex are all moved to this point.

λα= λ≡

Either of these (they are completely identical) causes a word inside RAID to have λ (not the contents of λ, but λ itself) deposited into it, and that location to be displayed on the screen with an arrow. The location is displayed in octal mode, but the mode can be changed, as with any other location displayed. This command does not move dot. This, then, is a way of seeing what some value is in other modes. For example, you might want to see what the octal value of the label FOO is (say FOOα=), or what the symbolic representation of 346 might be.

DEPOSITING IN CELLS λcr λ> λ< λα> λα< λβ< λβ> λε< λε> βE λβE

The only cell into which a value may be deposited is the one with the dot. Many commands move the arrow but not the dot, so they can become separated. Many commands can be prefaced by a value and will store that value in dot before the command is executed. This feature will be noted on each command that has it.

Note that FRAID (File RAID) in file-modify mode treats depositing in a cell by modifying the file on disk.

λcr λαcr λβcr λεcr
Note that all forms of carriage return are equivalent.

Carriage return, preceded by a value, causes that value to be placed in the location currently open, that is, the line with the dot. If the mode of the first or only part of λ typed can be recognized by the input format as text, Decimal, Octal, Real or Symbolic, λ will be displayed in that mode, no matter what the current mode settings are. You may then change the mode.

λ> λlf
λ< λ\
λα> λαlf
λα< λα\
λβ> λβlf
λβ< λβ\
λε> λεlf
λε< λε\
Note that linefeed is always equivalent to >, and \ is always equivalent to <. This is true for all combinations of control keys.

These commands have the same function as the equivalent forms without λ, except that the λ is first placed in the location pointed to, as with carriage return.

βE
λβE
The contents of λ are converted to text as if for displaying. If λ is omitted, "." is assumed. The mode used is λ's current screen mode if λ is on the screen; otherwise the current display mode. This text is sent to the system line editor, allowing you to edit it using the standard line editor commands. This text can then be activated with a carriage return, in which case the resulting value will be deposited in dot. (See the descripton of the carriage-return command for details.)

Note that this can lead to unfortunate results, since many of the display modes are incompatible with their associated input modes. For example, octal mode has a space between the two halves. Byte pointer mode (Q mode) has the word POINT. Sometimes what is loaded is NOT the mode displayed; this is true, for example of V mode (byte mode). So use this command with caution.

Once the text is in the line editor, special characters like ; and < will not activate, but they may be used anyway, followed by a carriage return to activate them. The carriage return will be interpreted as the "refresh screen" command, and won't hurt anything. Any character that normally causes activation in the line editor (like εð, where ð is any character) will always cause activation.

OPENING INDIRECT LOCATIONS tb λtb sm α[α] α@ β[β] β@

tb
λtb

Tab causes the location whose address is in the right half of the word currently pointed to by the right arrow to be opened with both the dot and the arrow. Note that if tab is preceded by a value, that value is first placed in the location pointed to, and THEN the location indicated by the right half OF THAT VALUE is opened. Tab is identical in action to β[and λβ[below.

;

A semicolon, with no control bits and not preceded by a value, causes the location indicated by the right half of the word currently pointed to by the right arrow to be displayed, as with tab, except that dot stays behind in the old place. Since dot is not changed, if something is deposited, it is deposited in the old location (where the dot is). Note, however, that if a further semicolon is typed (by itself), the location opened will be the one indicated by the right half of the new word, the word pointed to by the arrow. Note also that typing this command preceded by a value converts it to a different command; see above. Semicolon is identical in action to α[below.

α[
α]
α@
λα[
λα@
λα]

Each of these displays a new location based on the contents of the location currently marked by the arrow. The arrow is moved to the new location, but the dot is not changed. [uses the right half of the contents as its address,] uses the left half, and @ uses the effective address. If a value is used, it is deposited in the location pointed to by dot before the new location is opened, so if dot and arrow are on the same line, the new value will be used to calculate which location to display.

β[
β]
β@
λβ[
λβ@
λβ]

Each of these opens a new location based on the contents of the location currently marked by the arrow. Both the arrow and dot are moved to the new location. [uses the right half of the contents as its address,] uses the left half, and @ uses the effective address. If a value is used, it is deposited in the location pointed to by dot before the new location is opened, so if dot and arrow are on the same line, the new value will be used to calculate which location to display.

DYNAMIC LOCATIONS ϵ[ϵ] ϵ@ λϵ[λϵ] λϵ@

ϵ[
ϵ]
ϵ@
λϵ[
λϵ@
λϵ]

This command is a bit complicated. It causes the location currently pointed to by the arrow to be protected, then opens and protects the location addressed by that location. As usual, [uses the right half as a pointer,] uses the left half, and @ the effective address. However, if the value of the first location changes, the display will be updated to show the new location that the first location points to.

Suppose you wanted to keep track of the top location of the stack. You would say P; which would cause the push-down pointer (which everyone but DEC calls P) to be displayed (and pointed to). Then you would say ϵ[. This would cause the P location to be protected, then would open the location addressed by the right half of P, and cause it to be protected. However, from then on, whenever P changed, the location you just displayed would change to be the location addressed by the right half of P, that is, the top location of the stack.

A few special things are true of the dynamic location opened by this command. First of all, you should avoid putting the dot on this location on the screen and trying to deposit in it. You will wind up depositing in a table inside RAID, and your screen will do funny things. You are somewhat protected by the fact the location displayed by this command is marked only with the arrow, not the dot, so the only way of pointing to it is with the α> command and its friends. If you open this same location by normal means, it will be opened on another place on the screen, and you can deposit in it there. If in the P example above, the right half of P addressed location 300, and you said 300; , you would get location 300 opened in two places on the screen. The old one would be the dynamic one, and the new one would be an ordinary one.

It is possible to "chain" this command with itself (in any combination with the [,], and @ variants), for as many locations as there is room on the screen. If in the P example above, you had said ϵ[twice, instead of once, you would have opened the location addressed by the right half of the word on the top of the stack. This is all dynamic -- now to two levels -- so that if P changes, you have displayed, not only the new thing that P points to, but also the new thing that the new top of the stack points to.

The way to undo the dynamic chaining is to unprotect all the special locations on the screen; they will eventually be reused for something else.

Note, finally, that if this command is preceded by a value, that value is deposited in the location currently pointed to (by the dot), as above, BEFORE the operation takes place.

SEARCHES λαW λαN λαE ∨ ∧ λU λαU λΠ λαΠ

λαW

This is the word-search command. The general effect is to find all words that have λ in them. Specifically, RAID searches core, between certain limits, which may be set (see below); the initial bounds run over the whole core image except for locations 0-140 and the symbol table (but include RAID), as far as I can tell. The search is for words that match λ (not the contents of λ, but λ itself) in all bit positions that are on in location $M. That is to say, two words are compared by XORing them, and then ANDing the result with location $M. If this produces 0, the words match. RAID continues to search, opening each location that matches, until it comes to the end of the range, or until it has found enough matches to half-fill the display locations available, at which time it stops and prints a big star (*) on your screen. You may, at this point, type ∨ (the "or" key) to continue the search, and RAID will pick up where it left off, stopping when it has again half-filled the screen; or you may type any other command. (No characters are lost.) However, if you do not let a search run to completion, the next search you do will take up where the last one left off. You can, at any time later, type ∨ and continue the last search you did. If the search comes to an end, then OK will be displayed instead of a star.

λαN

This is not-word search. This works exactly like word search, except that words are considered to match only if they are different in some bit that is on in $M.

λαE

This is effective address search. It works like word-search except that for each word examined, the effective address is calculated, and this effective address is matched with λ. The mask in $M is not consulted, and words are considered to match if λ and the effective address are exactly the same.

∨ ∧ ("or" or "and"; they are equivalent)
This causes the last search you did to be continued. If you have done no searches, or if the last search you did has already run to completion, this command does nothing.

λU
λΠ

This causes the lower (λU) or upper (λΠ) search bound, for the next search only, to be set to λ. At the completion of the next search, this bound will be set back to its original value. Activate these commands with a carriage return.

λαU
λαΠ

This causes the lower (λαU) or upper (λαΠ) search bound to be set permanently to λ. This sets the value to which the search bound will be reset at the completion of a search.

PROGRAM CONTROL λαG αG λαB λβB αP λαP βP

The following section describes the RAID commands that allow you to run your program in various ways. These include the commands for manipulating breakpoints, which cause your program to pause when it gets to selected places, so that you can poke at it and see what's wrong. There are also single step features that allow you to run your program one instruction at a time while displaying important locations. None of these commands is legal in FRAID (File RAID).

Associated with several commands (including the searches above), is a big star (*), whicd RAID prints on your screen to let you know it is done with something that may have taken it a while. This star, in all cases, is removed the next time RAID recieves input -- usually as soon as you type anything.

λαG

This is the go command. It causes RAID to start running your program at location λ. RAID actually transfers to your program. Your program will continue to run until it hits a breakpoint, you type Call, or your program exits or does something the system doesn't like.

αG

The go command, without a value, starts your program at its starting address, i.e. the address in the right half of JOBSA, which is location 120.

λαB

This causes RAID to plant a breakpoint at location λ in your program. What RAID actually does, is change this location to a JSR to a certain location in RAID, remembering what the location used to be. This means that when your program gets here, it will JSR to RAID, at which point RAID will put the location back to what it was, open the break-point location on the screen marked with dot, arrow, and ex, and print a big star on the screen. You are now in RAID and can type commands to it.

λβB

This removes the break-point, if any, at location λ.

βB

This removes all break-points.

αP

This causes RAID to continue running your program from where it left off. If your program hit a breakpoint, RAID will continue your program with the breakpoint instruction (executing the real instruction there), and your program will run until it hits another (or the same) break-point, etc. If you have been stepping your program (see below), RAID starts it up at the next location to be executed. Don't use this command to start up your program the first time; use αG instead.

λαP

This causes RAID to procede (as above) from the current break-point (the last one you hit), λ times. That is, it has the effect of saying αP, λ times, as long as you hit the same breakpoint each time. If you hit other breakpoints in between, you will stop, then if you procede from them and hit the old breakpoint, the counting will continue. See the appendix on

breakpoints for details of getting out of this.

βP

This instruction places a temporary breakpoint at dot, and then proceeds as if you had typed αP. This breakpoint is removed as soon as you have stopped there once. Any value λ given with this instruction is ignored. There is no restriction on the number of temporary breakpoints in existence, except that the total number of breakpoints of any kind cannot exceed =16.

STEPPING, EXECUTING αS βS αX βX ϵX ϵY ϵE

αS

This is the basic step command. It causes the next location in your program (the next location to be executed; the one with ex) to be stepped. This means that the instruction has its effect, and then you are back in RAID. It is as if you had planted a break-point on the next instruction you would get to, and proceded. After stepping, RAID opens the next location (the next one you will execute) marked with dot, arrow, and ex. It does not print a star. If the instruction you step skips one instruction, RAID also displays the instruction skipped. This command has a different meaning in FRAID; it simulates setting $IO to -1, thus causing I/O instructions to be typed out.

βS

This is exactly like αS except that instead of stepping the next instruction of your program, it steps the instruction currently pointed to by dot. It then opens the location that that gets you to, and it becomes the next location to be executed. The dot, arrow, and ex all appear on this new location. Note that this is a way of getting started with stepping, if you haven't run any of your program yet, or if you want to change the flow of your program. This command has a different meaning in FRAID; it simulates setting $IO to 0, thus causing I/O instructions not to be typed out.

αX

This is the basic execute instruction. It has the same effect as αS, except if the instruction to be stepped (executed), is a subroutine call instruction (JSR, PUSHJ, JSA, or JSP), or a user UUO. In these cases, it treats the instruction and the subroutine (or UUO routine) it calls as one instruction. This means that your program starts running at the subroutine call (or UUO), and runs until it returns, and stops on the instruction it returns to. This instruction is then opened with dot, arrow, and ex. Note that if you STEP a user UUO, you wind up inside your UUO routines. There is a restriction involved in this command, and it applies also to the next two commands (the other two X commands). The restriction concerns how many locations a subroutine (or UUO) may skip. The maximum is 7. If you execute a subroutine call, or a UUO, (currently no system UUO skips more than 1, except INIT, which is handled as a special case by RAID), and it skips more than 7 locations, you will wind up in a funny place in RAID and all sorts of wrong things will happen.

A few words should be said about break-points in executed subroutines. In general, they work. You may hit a break-point inside a subroutine, the call to which was executed, and you may then step and execute other instructions. When you procede from the breakpoint, you will get back when the subroutine exits, just as if you hadn't hit any breakpoints. You should NOT step the subroutine return, as you will wind up stepping locations inside RAID. If you do this accidently, and haven't gone too far, you may procede (αP) and the right thing will happen. You may nest executes to a level of 8. You should avoid executing subroutine calls which you don't return from, as you will remain inside the subroutine, as far as RAID is concerned, until you do return, and this will decrease the number of levels you can nest subroutines.

βX

This works just like αX except that it starts with the

instruction currently pointed to by dot.

λϵX

This causes the instruction λ (λ itself, not the instruction at λ) to be executed as though it was in your program. Executing the instruction has no effect on which instruction is the next instruction to be executed, even if λ is a jump or skip. λ may be a subroutine call, in which case the right things happen. The restrictions listed above for executing subroutine calls apply. Note that executing a JRST with this command has no effect (except possibly on flags). The number of instructions skipped, if any, will be displayed at the top of the screen: "SKIP 2", for example.

λϵY

This has the same effect as ϵX if λ is a subroutine call. Otherwise, the instruction is just plain (vanilla) executed, regardless of what it is. Even jumps are not interpreted. If the instruction does not jump, it should not skip more than two, and control reverts to RAID as with ϵX. If the instruction does jump, you are off and running, as with αG, until you hit a breakpoint or something. The purpose of this instruction is to augment the αG command. Its principal utility is for saving a DMP copy of your program. You could type Call and then save it, but that would leave RAID in a funny state. You could type EXITϵX, but this would be like a subroutine that never returns, decreasing the number of levels available for subroutine nestings. EXITϵY will cleanly get you to the monitor so you can save the core image.

ϵE

This is like typing EXITϵY at RAID, except that all files that are open are closed. This exits RAID back to the monitor in such a way that the core image may be correctly saved and retrieved later. This command is legal from FRAID (unlike EXITϵY), and it releases the file after finishing all modifications.

MULTI-STEP ϵS λϵS X S

ϵS

This is the multi-step command. It has, except as noted below, the same effect as repeatedly saying αS. It steps the current location, updates the screen (displaying the next instruction to be executed), steps the next instruction, updates the screen, and so on. It keeps running until either you type a key, in which case it stops and returns control to RAID (the character you type may be anything, and is ignored); or it gets to a subroutine call or subroutine return instruction. PUSHJ, JSR, and UUOs are treated as subroutine calls. Subroutine return instructions are: POPJ, JRA, and JRST @. When it reaches one of these, it pauses and displays a big star.

If you type S it steps the instruction; if you type X it executes it. (If it is a subroutine call, this makes a difference.) It then procedes with multi-stepping. There are several other responses: If you type αS, then RAID will no longer stop on subroutine calls, but will always step them. If you type βS, then RAID will no longer stop at subroutine returns. If you type ϵS, then both things happen. If you type αX, then RAID will no longer stop at subroutine call, but will always execute them. If you type βX, RAID will no longer stop at subroutine returns. If you type ϵX, both things happen. This state of affairs remains in effect until you stop the stepping, or change it in the same way you set it, except that it is cleared at the start of each new multi-step command.

The multi-step mode can be terminated at any time by hitting some other character (the space key is recommened), which is then ignored.

λϵS

If a value is specified to the ϵS command then that many instructions are executed before doing the ϵS, unless a decision of the type mentioned above is required.

MACROS λαMδ λβM ϵδ

RAID has a facility for storing and executing macros, that is, sequences of commands (stored as characters) which you might want to invoke often, and would rather not type each time.

λαMδ

The value λ should be the address of the first word of an ASCIZ string which is to constitute the macro body. If λ is a byte pointer, then it is assumed that it points at the character before the first one. Recall that altmodes can be used to simulate control bits in RAID; this is useful if you want to put commands like ϵH in the string.

This command causes δ (which may be any single character, with or without control bits) to be defined as a macro with the body that λ points to. If δ is some character such that ϵδ already has some meaning, this command will replace that old meaning with the new one. Digits are good to use for macro names. Some characters, like the comma, will not work at all.

ϵδ

This command invokes the macro which has been given the name δ, which may be any single character. Any λ on this command is ignored.

λβM

This command invokes the macro whose address (as for λαMδ, above) is in λ without assigning it any particular name.

SHORT SUMMARY OF RAID COMMANDS
E = effective address, W = whole word, RH = right half,
@ = indirect, α = control, β = meta, ϵ = control-meta, ∞ = any bucky
λ = required argument, π = optional argument, ∂ = any character
(∂) = default for optional argument
pointers on the display: . → x

SCREEN CONTROL
παI -- # SCREEN ENTRIES = λ OR CUR+1
βI -- ZERO ALL SCREEN ENTRIES
ϵI -- ZERO UNPROTECTED SCREEN ENTRIES
∞cr-- REFRESH SCREEN

DISPLAY MODES
∞C -- CYMBOLIC MODE
∞O -- OCTAL MODE
∞D -- DECIMAL MODE
∞F -- FLOATING POINT MODE
∞H -- HALF-WORD MODE
π∞T -- CHARACTER MODE (λ(7)=7 => ASC, 6 => SIXB, 5 => RAD5Ø)
∞Q -- BYTE POINTER MODE
λ∞V -- BYTE MODE (λ=Ø => byte mask in $M+1, otherwise λ bits)
∞A -- ABSOLUTE MODE
π∞U -- E AS CHARACTER MODE
π∞J -- W AS FLAG MODE IN TABLE λ (Ø)
π∞R -- RH AS RIGHT-HALF FLAG MODE IN TABLE λ (Ø)
π∞L -- RH AS LEFT-HALF FLAG MODE IN TABLE λ (Ø)

SYMBOL TABLE COMMANDS
λα: -- NEW PROGRAM NAME λ
λα& -- NEW BLOCK NAME λ
παZ -- OPEN RECORDED BLOCK λ (1) BACK
βZ -- ZERO ALL BLOCK RECORDS
λ: -- DEFINE SYMBOL λ = .
λ←λ -- DEFINE SYMBOL=ADDRESS (λ1=λ2)
λαK -- HALF-KILL SYMBOL λ
λϵ: -- REVIVE HALF-KILLED SYMBOL λ
λβK -- ANNIHILATE SYMBOL λ

OPENING CELLS
λ; -- . TO λ
λ∞; -- . TO λ
πα; -- FREEZE λ (.)
λβ; -- UNFREEZE λ
λα= -- DISPLAY VALUE OF λ
λ≡ -- DISPLAY VALUE OF λ
α← -- . → TO x
LF -- ALWAYS EQUIVALENT TO >
 \ -- ALWAYS EQUIVALENT TO <
 > -- . → TO .+1
 < -- . → TO .-1
α< -- . → UP ONE ON SCREEN
α> -- . → DOWN ONE ON SCREEN
β> -- . → TO .+1 IN MODE OF .
β< -- . → TO .-1 IN MODE OF .
ϵ> -- . → UP ONE ON SCREEN IN MODE OF .
ϵ< -- . → DOWN ONE ON SCREEN IN MODE OF .

DEPOSITING IN CELLS
λcr -- DEPOSIT λ IN .
λ∞cr -- DEPOSIT λ IN .
λ∞> -- DEPOSIT λ IN ., THEN DO ∞>

λ∞< -- DEPOSIT λ IN ., THEN DO ∞<
λ∞[-- DEPOSIT λ IN ., THEN DO ∞[
λ∞] -- DEPOSIT λ IN ., THEN DO ∞]
λ∞@ -- DEPOSIT λ IN ., THEN DO ∞@
λTB -- DEPOSIT λ IN ., THEN DO TB
βE -- EDIT .

OPENING INDIRECT LOCATIONS
α[-- → TO @R(→)
α] -- → TO @L(→)
α@ -- → TO @E(→)
β[-- . → TO @R(→)
β] -- . → TO @L(→)
β@ -- . → TO @E(→)
 ; -- → TO @R(→)
TB -- . → TO @R(→)

DYNAMIC LOCATIONS
π∈@ -- FREEZE →, @E(→)
π∈[-- FREEZE →, @R(→)
π∈] -- FREEZE →, @L(→)

SEARCHES
λαW -- SEARCH FOR WORD λ
λαE -- SEARCH FOR EFFECTIVE ADDRESS OF λ
λαN -- SEARCH FOR ¬WORD λ
∨ -- CONTINUE SEARCH
∧ -- CONTINUE SEARCH
λU -- SET TEMP LOWER SEARCH BOUND
λαU -- PERMANENT LOWER SEARCH BOUND
λ∩ -- TEMP UPPER SEARCH BOUND
λα∩ -- PERMANENT UPPER SEARCH BOUND

PROGRAM CONTROL (NOT IN FRAID)
π∞G -- RELEASE CONTROL AT λ (@JOBSA)
λαB -- PLANT BREAKPOINT AT λ
λβB -- REMOVE BREAKPOINT AT λ
βP -- PROCEED TO TEMP BREAKPOINT AT .
παP -- PROCEED TO NEXT BREAKPOINT, REPEAT λ (0) TIMES

STEPPING, EXECUTING (NOT IN FRAID)
αS -- SINGLE-STEP ONCE FROM ×
αX -- SINGLE-EXECUTE ONCE FROM ×
βS -- SINGLE-STEP ONCE FROM .
βX -- SINGLE-EXECUTION ONCE FROM .
λ∈X -- EXECUTE SIMULATING INSTRUCTION λ
λ∈Y -- EXECUTE DIRECT INSTRUCTION λ

MULTI-STEP (NOT IN FRAID)
∈S -- MULTIPLE-STEP
λ∈S -- STEP λ TIMES THEN ∈S
 S -- STEP THIS JUMP
 X -- EXECUTE THIS JUMP
 ∂ -- UNLESS ∞S OR ∞X, TERMINATE MULTI-STEP

EXIT
∈E -- EXIT FROM RAID BY SIMULATING EXIT∈Y

FRAID SPECIAL COMMANDS
αS -- $IO ← -1
βS -- $IO ← 0

MACROS
λαMð -- DEFINE MACRO AT λ CALLED ð
λβM -- EXECUTE MACRO AT λ
ϵð -- EXECUTE MACRO CALLED ð

APPENDIX: SYMBOLIC MODE, PRINTING SYMBOLS, BYTE SIZE 0

SYMBOLIC MODE

This describes how symbolic mode decides how to display a word. If the left half of the word is 0, it is displayed in halfword mode. If the left half is all ones, the word is printed as a negative number. If the left-hand nine bits (opcode) is 0 or 777, the word is printed in halfword mode. Otherwise, if there is an entry for the opcode, the word is printed as an instruction. If there is no opcode entry, an opdef entry is searched for, and if that fails, perhaps it is a UUO. If none of these works, the word printed in half-word mode.

If the word is printed as an instruction, the index and accumulator fields are printed as symbols only if there is an exact match, otherwise as numbers.

PRINTING SYMBOLS

When RAID is going to print a number, unless it is in absolute mode, or some mode where numbers are always printed as numbers, it tries to print the number as a symbol, plus or minus an offset. To do this, RAID first searches the symbol table for the two best matches with the number it has, one greater, the other less. If the number is less than 140, RAID requires an exact match, or it prints it as a number. Otherwise, if it has found an exact match, it prints that symbol. If not, it goes through some contortions to decide which close match to use, and whether or not it will use either. There are four parameters it uses in deciding. These parameters are stored starting at location $C. The first parameter, the one at $C, is the maximum plus offset. The second one, at $C+1, is the maximum minus offset. Both of these numbers start out at 77, but may be changed. The value is anded with 777 before use. The third parameter we will call S, and is initially 10, the fourth parameter we will call Q, and is initially 40. We will call the plus offset P, and the minus offset M. RAID first compares P and M with their respective maxima. If both are too big, the number is printed as a number. If P is too big but M is not, then M is used (the minus one). If M is too large, but P is not, then P is used. If both are within the limits, then the fuction F=((P*Q)/100)-S-M is calculated, where 100 is octal. If F is positive, M is used, otherwise, P is used. This means that S and Q are relative weighting parameters; S is additive weighting, and Q is multiplicative. Notice that if Q=100 and S=0, RAID uses the smaller of M and P. If Q is instead 40, P is used unless it is twice as big as M. On the other hand, if Q=100, but S=10, P is used unless it is greater that M by more than 10.

BYTE SIZE 0

If the byte size in the V mode command, or in the α% input string is 0, the bytes are interpreted acording to a mask in location $M+1. Bytes may be any sizes, and the boundaries are indicated by a change from 0's to 1's, or vice-versa, in this mask. For example, if $M+1 contains 707070707070, this would indicate 3-bit bytes. 770077007700 would indicate 6-bit bytes. 741703607417 would indicate 4-bit bytes. 770770770770 would indicate a 6-bit byte, followed by a 3-bit byte, followed by a 6-bit byte, followed by a 3-bit byte, etc. 252525000000 would indicate 18 one-bit bytes followed by a 18-bit byte.

APPENDIX: RAID DEFINED LOCATIONS AND TABLE OF POINTERS

These are all global symbols. You can declare them EXTERNAL, then write clever programs to poke at them. Note that in this context, $ refers to a dollar-sign, not an altmode.

DDT
: This is the starting address of RAID.

DDTEND
: This is the first unused location after RAID. If you have a program without RAID which runs into trouble at user location 12345, say, then you can find the offending instruction by loading the program with RAID and examining location 12345+DDTEND-140.

$C
: This and the three locations following it are the parameters for deciding how to print symbols. See above.

$M
: This location is the search mask. It is initially set to -1. See the search commands.

$M+1
: This is the byte mask for 0 byte size. See above.

$M+3
: This is the flag name table pointer. See section on the J display mode.

$IO
: If non-zero, words which represent machine I/O instructions (CONI, DATAO, BLKO, etc.) will be printed as such in symbolic mode. Otherwise they will be treated as the UUO's PPIOT, MAIL, INTUUO, etc., which share some of the same Opcodes, or as simply negative numbers if no such UUO exists for this word. Changing this location has no effect on words currently being displayed; to see what a displayed location has under this mode, switch it to octal and back to symbolic. FRAID has the commands αS and βS to set this cell to -1 and 0, respectively.

$I
: This is the location where RAID keeps its current idea of your program counter -- the address of the next instruction to be executed. Breakpoints JSR to this location. The left half contains your program flags. If you change the left half of this word, you will change what program flags get restored each time your program is started up.

$1B to $20B
: These 20 locations, and the four locations following each of them, are the breakpoint table. For a detailed description of what each contains, see the appendix on breakpoints. The first word is the address of the breakpoint. The location is -1 if this breakpoint is unused. The second location is the multiple procede count. The third location is the conditional skip instruction. The fourth location is string-breakpoint byte pointer. The fifth location is the real contents of the breakpoint location.

There is a table just before DDT of pointers to useful entities in Raid:

```
DDT-1    address of $SBP routine
DDT-2    address of $RBP routine
DDT-3    address of $M
DDT-4    address of $IO
DDT-5    address of $1B
DDT-6    address of $I
DDT-7    address of $C
DDT-10   address of DDTEND
DDT-11   address of $BGDDT
DDT-12   address of $RPTCNT
```

These are useful for programs which want to avoid undefined globals and runtime testing of availability, if Raid is not loaded.

The lefthalf of JOBDDT is 40 (version 1).

APPENDIX: BREAKPOINTS

You may have a maximum of 20 (octal) breakpoints at any one time. The breakpoint information is contained in a table, with five locations for each breakpoint. The first location of each of the 20 entries is given a label of the form $nB, where n is a number from 1 to 20. Entries are assigned to breakpoints in the order in which breakpoints are created, starting with $1B.

This first location of each entry contains the address of the breakpoint in your core. It is -1 if this breakpoint is unused. The left half is non-zero (40,, bit on) if the breakpoint is temporary.

The second location in each entry ($nB+1), contains the multiple procede count for that breakpoint. This is where RAID puts the count if you say λαP. This count is counted down by 1 each time you hit this breakpoint and the breakpoint is ignored (you procede automatically) if the count is still positive. Depositing a number here will have the same effect as using multiple procede. Depositing 0 here will get you out of a multiple procede.

The third word of each entry is the skip instruction. If this instruction is non-zero when RAID hits this breakpoint, RAID executes the instruction (which may be a subroutine call), and what RAID does with the breakpoint depends on whether or not this instruction skips.

skip 0: normal
If the instruction does not skip, RAID does the normal thing (what it would have done if this word had been 0), namely, it counts the multiple procede count and procedes if it is positive, stops if it is zero or negative.

skip 1: stop
If the instruction skips once, RAID does not count the multiple procede count, but rather it stops at the breakpoint anyway.

skip 2: procede
If the instruction skips twice, RAID does not count the multiple procede count, but rather it procedes (ignores the breakpoint), anyway.

The fourth location in each entry ($nB+3) is the string breakpoint pointer. If it is not zero, then it is assumed that the right half points to (addresses) the start of an ASCIZ string. The left half, if non-zero, implies that the word is a byte pointer such that ILDB on it will fetch the first string character. If it's zero, a byte pointer to the first character in the word will be constructed. This ASCIZ string is then scanned by the input scanner, just as if you were typing those characters on your keyboard, every time you stop at this breakpoint. This means that if you want to display a certain location each time you hit a certain breakpoint, you can put in the appropriate location a pointer to an ASCIZ string consisting of FOO; (to open location FOO). When the string runs out, RAID takes input from the keyboard. You can use the α" feature to create long strings, if you need them. Instead of using control bits, use the altmode equivalents.

The real contents of the breakpoint location are stored in the fifth of the locations in the table entry (at $nB+4). Changing this location, however, has no effect, because the real contents are replaced in your core while you are talking to RAID, and the JSR is

placed there only while your program is running.

If a program that is loaded with Raid executes the instruction JSR AC,$SBP ($SBP is a global defined in Raid), a breakpoint will be set at the location specified in AC. There is no way to specify the count, skip instruction or string yet. Executing a JSR AC,$RBP will cause any breakpoint set at the location specified in AC to be removed. Pointers to $SBP and $RBP can be found in the table immediately preceding DDT.

A program may simulate hitting a breakpoint by executing JSR $I ($I is a global defined in Raid). You may then procede from this breakpoint in the usual way. When Raid is entered by transferring to location DDT (or RAID), Raid simulates a breakpoint from the contents of JOBOPC. Thus if you type <call>DD<cr> at your program, you may procede it by typing αP at Raid.