TRANSMITTAL NOTICE

Date: May 1991

Publication Number: CSM-1023-000

Publication Title: DMS Macrocode Assembler Programmer Reference Manual

Remove	Insert	Page Numbers	# of Sheets	General contents of manual described and changes (if any) listed
	★	All	13	May 1991. Original printing. This document describes the syntax for writing programs with the device maintenance system (DMS) assembler. The format of assembler input and output files, syntax, and the macrocode instruction set are described.

Cray Research, Inc.

Reader Comment Form

Title: DMS Macrocode Assembler Programmer Reference Manual **Number:** CSM-1023-000

Your feedback on this publication will help us provide better documentation in the future. Please take a moment to answer the few questions below.

For what purpose did you primarily use this manual?

_____Troubleshooting
_____Tutorial or introduction
_____Reference information
_____Classroom use
_____Other - please explain _____

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria and explain your ratings:

_____Accuracy _____
_____Organization _____
_____Readability _____
_____Physical qualities (binding, printing, page layout) _____
_____Amount of diagrams and photos _____
_____Quality of diagrams and photos _____

Completeness (Check one)

_____Too much information _____
_____Too little information _____
_____Just the right amount of information

Your comments help Hardware Publications and Training improve the quality and usefulness of your publications. Please use the space provided below to share your comments with us. When possible, please give specific page and paragraph references. We will respond to your comments in writing within 48 hours.

NAME_____
JOB TITLE_____
FIRM_____
ADDRESS_____
CITY_____STATE_____ZIP_____
DATE_____

[or attach your business card]

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attn: Hardware Publications & Training
770 Industrial Boulevard
Chippewa Falls, WI 54729

Fold

STAPLE

Record of Revision

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version, and the new version is assigned an alphabetic level which is indicated in the publication number on each page of the manual.

Changes to part of a page are indicated by a change bar in the margin directly opposite the change. A change bar in the footer indicates that most, if not all, of the page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

REVISION	DESCRIPTION
	May 1991. Original printing.

Preface

This programmer reference manual is written for programmers, field engineers, and anyone requiring detailed information on the device maintenance system (DMS) assembler, specifically.

Topics covered include invoking the assembler, file formats, syntax, variable space allocation, and assembly errors. An instruction reference table is also provided. The information contained in this manual does not duplicate information contained in the *IOS Model E Offline Diagnostic Reference Manual*, which is used for broader reference purposes in the field and in STCO. Refer to the *IOS Model E Offline Diagnostic Reference Manual*, publication number CDM-1018-000, for information on DMS, DME, MOS, tests, utilities, and monitors.

The following conventions are used throughout this manual:

- Square brackets [] indicate an optional entry.
- Angle brackets < > indicate a required entry.
- **Bold** type indicates a command as discussed in text.

Reader comment forms are located at the front and the back of this manual. Please use them to offer comments, suggestions, or corrections regarding this publication. Information on other hardware and software publications can be obtained via the online publications catalog.

Contents

Assembler Syntax (continued)

Tables

DMS Macrocode Assembler

This document describes the syntax for writing programs with the device maintenance system (DMS) assembler, referred to in this document as *dmsasm* or simply as *the assembler*. The format of assembler input and output files, syntax, and the macrocode instruction set are described.

Invoking dmsasm

The assembler resides in the system binary directory along with all other Cray Research MWS application programs. The assembler is invoked from the UNIX shell with the following format (square brackets [] indicate an optional entry, and angle brackets < > indicate a required entry). The only required argument is the name of the assembly source file, <source.d>.

```
dmsasm [-i <incl>] [-o <outfile>] [-n <lpp>] [-D <sym> [=value]] [-w <limit>]
[-lvu] <source.d>
```

-i <incl>	Includes the device module specified by <incl> in the assembly. Has the same effect as the line *include incl* in the source file. You may specify more than one -i option.
-o <outfile>	Gives the output binary file a name specified by <outfile> (instead of the default). The listing file also has the name <outfile>.lst.
-n <lpp>	Sets the number of lines per page in the listing to <lpp>. If this number is not specified, the assembler looks for the UNIX environment variable lPP and uses its value as the number of lines per page. If lPP does not exist, the default setting is 66.

-D <sym> [=value]	Defines the symbol <sym> as an equate as if it were defined in the source code. A value may be assigned by including the [=value] portion of the definition; the default value is 1 (true). The assignment string may have to be put in quotation marks to protect it from premature processing by the shell. Symbols defined in this way should only be used as arguments to IF pseudo-instructions.
-w <limit>	Instructs the assembler to issue a warning message if the code (excluding text displays) exceeds parcel limits. The default limit can also be set with MACROLIMIT = <limit> in the shell environment. (refer to "Environment" below).
-l	Disables listing file generation.
-u	Disables warnings about unreferenced variables.
-v	Causes the assembler to print out the version number when invoked. The version number is also indicated in the listing.

The assembler sets the return value that is passed to the shell (accessed as $?) to the error count for the assembly (0 means no errors).

Environment

The assembler uses two UNIX environment variables:

- LPP, if set, determines the decimal number of lines per page in the listing.
- MACROLIMIT, if set, determines the maximum code/variable size (in octal parcels) to allow before issuing a warning. If this variable is not set, no checking is done. The limit check is only performed on code and variable space, not on displays.

File Formats

There are four file types involved in using the assembler:

File	Description
Source files	ASCII text files read by the assembler.
Listing files	[optional] output listings from the assembler.
Binary files	Binary (macrocode) output from the assembler.
External device modules	Files containing instructions specific to a certain device (such as a disk drive), which are explicitly included in a program at assembly time.

Source Files

The assembler source files are simple ASCII text files, which may be created with the UNIX *vi* editor or any other text editor. Tabs and character spaces are treated equally. Where space is allowed in the syntax, any number of tabs and/or spaces may appear. Each line is always terminated by a UNIX newline character. The maximum allowed length of a source line is 80 characters. Long literal strings may also be continued (refer to "Literal Constants" on page 6).

Source file names should end with a .d suffix; however, this convention is optional and is neither enforced nor assumed by the assembler. Therefore, you may use any file name for the source file. The advantages of using the .d convention are easy generation of output and listing of file names. If you do not use a .d suffix, the output file defaults to the generic name dms.out. Refer to "Listing Files" and "Binary Files" for more information.

NOTE: Even if the file ends in a .d suffix, you must still type the full file name on the command line.

Listing Files

The assembler, by default, produces a listing from an assembly. The listing can be disabled with the -l command line option. The format of a listing line is shown below.

```
14:    000014 000063 000062 000027        10. b = a & b'10111
```

The maximum width of listing file lines is 132 characters. The elements of the listing line shown above are as follows:

- The octal parcel address of the instruction in the binary output followed by a colon. (14:)
- The octal 4-parcel macrocode instruction as it appears in the binary output. This field is blank for instructions that do not produce binary output, such as pseudo-instructions or comments. (000014 000063 000062 000027)
- The line number of the source line being assembled. This number should correspond to the line number in the source file. (10.)
- The actual input source line. (b = a & b'10111)

The source listing is followed by a listing of all defined program labels, variables, and equates. Variables are listed with their parcel addresses. Equates and labels are listed with their values. The listing file name is the name of the binary output file with the .lst suffix appended. If your input file does not end with .d, the listing has the default name dms.out.lst (unless overridden with the -o command line option).

Binary Files

The final output of the assembler is a pure binary file containing the macrocode generated by the assembler. The amount of code generated determines the length of the file. Only enough of the file is saved to contain the code; no trailing 0's are appended. The default name for the output file is the name of the input file with the .d removed, or if specified, the name given by the user with the -o command option. If the input file does not end with .d, the binary file is named dms.out.

External Device Modules

The assembler is designed to have only the universally available part of the macrocode instruction set available internally. Keep any device-specific instructions externally in separate files. Include the files at assembly time by using the INCLUDE pseudo-instruction in the program. The requirements for these external files are described in the following subsections.

File Location and Naming

The *include* files must reside in the dmssys directory, which must be a subdirectory of your home directory. Each file is named for the device it represents, followed with a .i suffix. For example, a file for a DD-40 disk device would be called DD40.i and must be in the $HOME/dmssys directory, where $HOME is the value of the user's UNIX HOME environment variable. This file could then be referenced with the statement *include "DD40"* in your source program.

NOTE: File names are case-sensitive (*include "dd40"* would not work in this example). You can avoid this problem by making multiple links with different names to the device file, so *dd40* and *DD40* reference the same file.

File Contents

The *device include* files contain a series of instruction name/opcode entries, one per line. Each line consists of three fields separated by one or more spaces. The first field is the ASCII mnemonic representation of the instruction, which is not case-sensitive. The second field is the octal representation of the instruction's opcode. The number requires no special formatting. The third field is a bit mask that determines which arguments may be used with this instruction. Bits 2, 1, and 0 correspond with the first, second, and third arguments, respectively; thus the number has a value from 0 to 7.

A 1 bit indicates that the argument is used; a 0 bit indicates that the argument cannot be used. A 0 bit causes an assembly error if an argument other than 0 appears in that field. A 1 bit issues a warning only if an unexpected argument is used. If the assembler detects an error in the include file while loading, an appropriate warning message is issued and the rest of the include file is discarded.

Assembler Syntax

This section describes the basic types of assembler source input: comments, constants, literal constants, identifiers, expressions, operators, variable addresses, statements, assignments, indirect assignments, jumps, pseudo-instructions, and text displays.

The input to the assembler is not case-sensitive except in literal strings. Any line may begin with a label that may be used in jump instructions. A label must begin with the first character of a line, or else there must be at least one space before the start of the line. More specific information is provided below.

Comments

A comment always begins with a semicolon (;). Comments may appear anywhere in a source line. Anything between the semicolon and the end of the line is ignored.

Constants

A constant in the assembler context is a number represented in one of four bases: octal (default), decimal, binary, or hexadecimal. Specify the base of a constant by prefixing it with the characters O' for octal, D' for decimal, B' for binary, or H' for hex (the letters can be lowercase). If you do not specify a base prefix, the number is interpreted in the current active base, set by the BASE pseudo-instruction. The EQU pseudo-instruction can also be used to define symbolic constants (equates).

Literal Constants

A literal constant is a text string enclosed by quotation marks (" "). Any literal constant can continue on two or more source lines if it exceeds the 80-character limit. To continue a line, end it with a backslash (\); any characters after the backslash on that line are ignored. Subsequent lines are treated as continuations up to the ending quotation mark. Continuing lines can be indented; any space at the beginning of continuing lines is ignored. The maximum length of a literal constant is 255 characters. Literal constants are used mainly in the INCLUDE, TEXT, PFMT, and TFMT pseudo-instructions.

Identifiers

In dmsasm, an identifier can refer to a label or a named variable. The rules for identifiers are as follows:

- Identifiers must begin with an alphabetic character (a-z or A-Z).
- Identifiers consist of up to 31 letters, digits, or underscore (_) characters.
- Identifiers are **not** case-sensitive; therefore, *Name* is the same as *name*.
- Identifiers are defined as labels if they begin with the first character of a source line, unless a line contains a CON pseudo-instruction, in which case the label is treated as a variable definition.
- Identifiers are **not** declared before use; they are defined by the first reference to them, as in the BASIC language.
- The user can allocate storage for variables within the program using the CON pseudo-instruction. Variables not defined this way have space allocated for them at the end of assembly by the assembler.
- Identifiers may not have the same name as a reserved word.
- All variables use 1 parcel of storage.

Expressions

There are two types of expressions recognized by the assembler: constant expressions and variable expressions. A constant expression is a grouping of numerical constants, symbolic constants (equates), and operators. A variable expression is a variable grouped with a variable or constant expression through a single operator. An expression containing two variables cannot contain constants, because having both requires more than one macrocode instruction. The constant expression is always grouped together and evaluated by the assembler before it is combined with the variable.

In instructions that take two arguments where one is a variable and one is a constant, the variable must be the first argument.

Refer to the "Instruction Reference" subsection for more details. Below are examples of code using expressions:

```
x=y+1                    ; OK
x=1+y                    ; illegal
100                      ; constant
100*2+34                 ; constant expression
(100 & B'101) >> 12      ; constant expression
var1 + 100               ; variable expression
var2 * (100 + 234)       ; variable expression
var2 * 100 + 234         ; equivalent to the above
var2 * 334               ; same result as above
var1 + var2              ; OK
(var1*100) + var2        ; Illegal!
temp  = var1 * 100       ; Correct implementation
final = temp + var2      ; of expression above
```

Operators

The operators are listed below. Some of them can only operate on constant expressions, while others can also operate on variables. All of the operators can be applied to constants; those marked with asterisks (*) can also be applied to variables.

Operator	Description
+	Addition*
-	Subtraction*
*	Multiplication
/	Division
%	MOD (remainder)
~	Logical complement (unary)
-	Unary minus (negative number)
&	Logical AND*
|	Logical OR*
^	Logical exclusive-OR*
>>	Logical shift right*
<<	Logical shift left*

All the operators except the unary operators (~ and -) have the same priority. Therefore, expressions are evaluated strictly from left to right. Parentheses may be used to change the order of evaluation. Unary operators have precedence over the others. For example:

```
123 << 12 + 15 * 27          = (((123<<12)+15)*27)
123 << ~12 + 15 * -27        = (((123<<(~12)+15)*(-27))
```

Variable Addresses

The @ operator allows a variable's address to be used in an expression. Refer to the code example in the "Indirect Assignments" subsection.

Statements

In general, a statement is a macrocode mnemonic followed by a space and the instruction arguments (if any):

```
[Label] <keyword> [arg1[,arg2[,arg3]]]
```

The number and meaning of each instruction's arguments are described in the instruction description and in the documentation for the macrocode interpreter. The arguments are separated by commas and spaces (optional); an instruction may have from zero to three arguments. An argument may be a constant, a label, a variable, or a constant expression. The first argument generally must be an address (of a variable) for the instruction to operate on. The second argument (optional) may be a variable address or constant data. The third argument may have a number of meanings depending on the instruction. Consult the "Instruction Reference" subsection for further information.

The assembler issues an error message and zeros out an instruction if an argument appears in a position where none is allowed. The assembler issues a warning if an argument does not appear where one is expected, but still generates code for that instruction. There is a one-to-one correspondence between the arguments' positions on the source line and their positions in the binary code. For example, the following line would assemble into 000200 000010 000000 000000 (assuming the opcode for SEEK is 200 octal):

```
seek    10
```

The following line would assemble into 000200 000010 000020 000030. Consult the definition of each instruction for the correct placement of arguments.

```
seek    10,20,30
```

Assignments

For increased readability, the assembler provides an alternate way of writing many simple operations, which looks more like a high-level language specification than an assembler. This syntax can substitute for any of the binary logical or arithmetic instructions supported by the interpreter. Assignments in dmsasm encompass all operations that store a result into a variable.

The general syntax is as follows:

```
[Label]  <var> = <const expr>
```

or:

```
[Label]  <var> = <var> <operator> <var or const>
```

The simplest assignment is to set a variable to a constant. Any of the variable operators listed previously may also be applied to a variable expression in an assignment. For an example of how the assembler rewrites an operation, refer to the following line, which is translated by the assembler from:

```
a = a + 100
```

to:

```
ADDL a,100,a
```

Indirect Assignments

Assignments can also be made indirectly where a variable's contents are used as an address of data to be retrieved. The indirect operand is enclosed in square brackets ([]). Indirect operands can appear on either side of an assignment, but not on both. Examples of code areas follow.

```
data    con       100        ;  declares data as a variable
        con       0
        a = data             ;  a = 100
        addr = @data         ;  addr = data's address
        b = [addr]           ;  b = 100 (data's contents)
        addr = addr+1        ;  increment address
        b = [addr]           ;  b = 0 (contents of [data+1])
        [addr] = 200         ;  store 200 at "data+1" location
        [a] = [addr]         ;  Illegal (src & dest both indirect)
```

You cannot mix expressions with indirect addressing because doing so requires more than one instruction.

Jumps

An alternate form for jump instructions is provided. The macrocode instruction set provides for both conditional and unconditional jumps. The general syntax for a jump is as follows:

```
[Label]  jump  address [, condition]
```

The conditions may be applied to a variable and a constant or to two variables. A conditional expression takes the following form:

```
<var> CONDITION <var or const expr>
```

The first part of a conditional comparison **must** be a variable, as defined by the interpreter syntax of the jump instructions. The valid conditions are as follows:

```
==          if expr1 equals expr2
!=          if expr1 not equal to expr2
>           if expr1 > expr2
<           if expr1 < expr2
>=          if expr1 >= expr2
<=          if expr1 <= expr2
```

Examples of code are as follows:

```
jump L1              ;  unconditional jump
jump L2, a > b       ;  jump if a > b
jump L3, x != 0      ;  jump if x non-zero
jump lab, x < 0      ;  jump if x is negative
```

The instructions above would be translated by the assembler into the equivalent assembler statements below:

```
JUMP      L1
IFGT      a,b,L2
IFNE      x,0,L3
IFLT      x,0,lab
```

Pseudo-instructions and Text Displays

The assembler pseudo-instructions and text displays are described in the following subsections. The syntax for a pseudo-instruction is as follows:

```
pseudo  [const expr or literal]
```

The general pseudo-instructions are BASE, BSSZ, CON, EQU, INCLUDE, IF, ELSE, ENDIF, and ENDASM. All may be entered in either uppercase or lowercase letters.

There is also a set of special pseudo-instructions used to set up APML-type diagnostic displays to be decoded by DMS; these are DISPLAY, DEFDISP, TEXT, PFMT, and TFMT. At the end of a macrocode program the programmer can include formatted text displays, which are available for viewing under DMS as the program executes. The displays correspond to function keys on the terminal as follows (only keys associated with user-defined displays are listed):

Key	Function
F1	Running display
F2	Error display
F3	Help display
F5	Device status display
F6	Parameter (test setup) display

Each function key can display multiple screens (up to 64). The user can scroll through the screens with the **pf** and **pb** commands in DMS. Corresponding to each display (defined by a DISPLAY pseudo-instruction) is fixed text and a list of formats that define memory locations displayed within the text.

BASE

The BASE instruction sets the active number base. The argument to BASE is a constant expression with values of 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). Any other numbers cause errors and are ignored. Any number of BASE instructions may be included in a program.

BSSZ

The BSSZ instruction reserves a block of memory cleared to 0's. The argument is the number of parcels to reserve, which is always rounded to a multiple of four.

CON

The CON instruction reserves 1 parcel of storage at the current location with the value specified in the CON statement inserted into it. If code is generated after a CON statement, the code address is automatically rounded to a multiple of four again.

EQU

The EQU instruction equates a symbol with a value. The form of the equate line is as follows:

```
<Ident> EQU <const expr>
```

This instruction defines the value of <Ident> to be the same as the constant expression argument. Unlike labels, equates must be defined before they are referenced or they produce erroneous results. An equated identifier may be used any where a numeric constant can be used.

INCLUDE

The INCLUDE instruction means to include a device module. The argument is a literal constant and the name of the device, which must correspond exactly to the name of the device file in the dmssys directory as described previously. The .i suffix must not be included in the name. An INCLUDE pseudo-instruction may appear anywhere in a program. The instructions defined by the include file are only recognized after the INCLUDE pseudo-instruction is assembled.

IF, ELSE, ENDIF

These instructions allow conditional assembly. The IF pseudo-instruction takes one argument, which must be a constant or constant expression. The expression may also contain relational operators ==, !=, >, <, >=, and <=. If an undefined symbolic name is used in the expression, it is treated as having a value of 0, which allows optional definition of symbols from the command line.

This expression is evaluated at assembly time and is treated as true if it is nonzero; if it is zero, it is treated as false. If true, the code following the IF instruction up to the matching ELSE or ENDIF instruction is assembled; if false, it is skipped. The ELSE instruction is optional, but each IF instruction must have a corresponding ENDIF instruction. Skipped code is not shown on the listing. IF blocks may be nested up to 32 deep.

ENDASM

The ENDASM instruction unconditionally terminates assembly of the source file, whether or not the end of the file is reached. The ENDASM pseudo-instruction is optional at the end of a program.

DISPLAY

The DISPLAY instruction and the DEFDISP, PFMT, and TFMT instructions, described below, generate APML-type text displays. They should only appear after all program code in the source file. There must be exactly one DISPLAY pseudo-instruction at the beginning of your display definition for the displays to be generated correctly. This instruction takes no arguments.

The first function of the DISPLAY pseudo-instruction is to force allocation of all program variables, which means that no new variables may be used after the DISPLAY instruction is assembled. You can, however, define new labels for text and format statements. The DISPLAY instruction also generates the T line for the display header (used by DMS to read the displays).

DEFDISP

The DISPLAY pseudo-instruction is immediately followed by one or more DEFDISP lines with the following format:

```
[Label]  DEFDISP  <display #>,<format addr>,<text addr>,
<scroll length>
```

This line follows the order used in the VWD statement used to define the MWS/CPU interface data in APML. The display number is the display designator recognized by DMS. The format and text addresses are normally labels assigned to the display text and formats, which follow the DEFDISP pseudo-instructions. The scroll length field is used to indicate to DMS that the display can be scrolled with the **df** and **db** commands, and to indicate to DMS how many parcels to scroll the display. If the scroll length field is set to 0, DMS does not allow scrolling on that display screen.

NOTE: The DEFDISP section must be followed by a parcel of all 1 bits (CON -1).

The first argument (<display #>) defines to which key the display corresponds from the choices listed below. Any other display number is invalid for user displays and is ignored by DMS. The range of numbers allows for the stated 64 (0-77 octal) displays per key. For multiple pages, include one DEFDISP line for each page with succeeding display numbers (for example, displays 200-203 define four pages of error display).

Key	Display Number (octal)
F1	100-177
F2	200-277
F3	400-477
F5	700-777
F6	500-577

DMS currently ignores the lowest 6 bits of the number and simply defines the displays in the order in which they are defined in the program, but this process is not guaranteed to be true in the future.

The second argument (<format addr>) is the address of the beginning of a list of PFMT and TFMT statements that defines the data to be displayed. This field is normally the label of the first format line.

The third argument (<text addr>) is the address of the start of a list of TEXT pseudo-instructions that defines the fixed text portion of the display.

The fourth argument (<scroll length>) defines the total length (highest address to lowest address) of a display so that DMS can reference multiple memory images using the same display. For example, if the program contains a series of error information buffers, each 10 parcels long, then a single error display screen can be used by DMS to display any one of the buffers by adding the scroll length as an index to the base address of the first display.

TEXT

The TEXT pseudo-instruction defines ASCII format text that can be used in programs, and ASCII format for display text and message lists (refer to "Message List"). The format of the text is the same as defined for APML text displays. The syntax is as follows:

```
[Label]  TEXT  <literal constant>
```

For displays, a list of TEXT pseudo-instructions is defined, one for each line of the display (20 lines maximum). The first line usually has a label used as the third argument of the DEFDISP pseudo-instruction.

NOTE: The text definitions must be followed by a parcel of all 1 bits (CON -1).

Each line must end with a tilde (~) character, which denotes the end of line to DMS. Thus, a line of text as displayed on the screen can be formed of more than one TEXT line, with the last line terminating with a tilde. It is recommended, however, that a one-to-one correspondence be maintained, and text line continuation (\) be used for long lines instead. Finally, if the line is terminated by a double tilde (~~), DMS clears from the end of the text to the edge of the screen; otherwise, the screen contents after the end of the text are not cleared.

PFMT and TFMT

These two pseudo-instructions put formatted data onto the screen at a specified position. The basic unit of display is a single parcel (or a single macrocode variable). The address of the parcel to display is normally the name of a variable or the label on a CON pseudo-instruction, although a numeric address can be hard-coded (not recommended because the address is taken relative to the DMS diagnostic base for the currently selected device).

PFMT is the equivalent of the APML parcel format macro. Place any options in a literal string, separated by commas and/or one or more spaces. Each option takes the form OPT = value, where OPT is one of the allowed options for PFMT, and value can be a constant, variable, label, or equate name. The option may be in uppercase or lowercase letters. The value of a numeric constant is interpreted according to the current base setting. The base can be overridden in the usual way, by preceding the number with the <base type> notation. If the base is hexadecimal, a number must begin with a digit. No spaces are allowed around the = character within options. The same default values for all options are, in effect, the same as in the APML macros.

To obtain a PFMT with just default options, use a null string surrounded by quotation marks as the argument. The PFMT syntax is as follows:

```
[label]  PFMT  "[opt=value][,opt=value][,...]"
```

The options allowed are sadd, line, col, np, pm, ndr, ndl, bbp, ebp, slz, xd, txt, bold, dec, and base. The base, dec, and bold options are unique to DMS.

The PFMT options are described below. Many of the options are flags with value 0 equaling false or 1 equaling true. Although all options have default values, the options sadd, line, and col should always be included.

Option	Meaning
sadd	Start address for display (variable name).
line	Line number on which to display (range 1-20).
col	Column number on which to display (range 1-80).
np	Number of parcels to display (default = 1).
pm	Parcel merge flag; if set, combine np parcels into one number (default = 0).
ndr	Number/digits right justified; specifies field width and that the number is right justified and filled with leading 0's on the left (default = 0 [use ndl instead]).

Option	Meaning
slz	Suppress leading zeros flag (default = 0).
ndl	Number/digits left justified; specifies field width and that the number is left justified (default ndl = 6).
bbp	Beginning bit position; allows extraction of a range of bits from the data (range 0-15, default = 0).
ebp	Ending bit position; used with bbp to extract a range of bits (range 0-15, default = 15).
xd	Allow extended digits; if the number does not fit within ndr/ndl digit limits, the field is extended to display the whole value. If xd = 0, the value is truncated instead (default = 0).
txt	Flag specifies to display the data in ASCII format (default = 0).
bold	Flag specifies to display the data in bold (reverse video) (default = 0).
dec	Flag specifies to display the data in decimal instead of octal (default = 0).
base	Flag signals DMS to display the data from the device's buffer area instead of from the diagnostic area (default = 0).

The TFMT pseudo-instruction is used in the same way as PFMT to display running text messages. The sadd field in a TFMT contains the address of a location that contains a message number. When a nonzero value is inserted into the sadd location by a running program, DMS displays the message associated with that number at the location given by the line and col options to TFMT (refer to "Message List"). The madd option allows numeric data to be inserted into a line of text. The data starting at the madd address is inserted into the text wherever one or more # characters appear (one # for each octal digit desired). Each successive # grouping is replaced with a number from the next parcel after the first madd address. None of the other PFMT options are valid for TFMT. The TFMT syntax is as follows:

```
[label]  TFMT  "[opt=value][,opt=value][,...]"
```

NOTE: The format list must be followed by a parcel of all 1 bits (CON -1).

Message List

Each macrocode program can only have one message list. This list consists of TEXT pseudo-instructions that define a list of running text messages for display with TFMT pseudo-instructions. Each message is terminated with the single or double tilde as is the display text. The messages are assigned numbers in the order they are defined, starting with 1 (with a maximum of 200 currently supported by DMS). When the program displays a message, it puts the message number into the variable listed as sadd in a TFMT pseudo-instruction, and the message is displayed. To erase the message, the TFMT location is written with a 0.

To define a message list, a DEFDISP pseudo-instruction is used with a display number of 1000 octal. The <format addr> and <scroll length> arguments are ignored; the <text addr> argument is the address of the start of the message list described above.

Variable Space Allocation

At the end of assembly, the assembler scans a list of variables defined in the program and sequentially allocates 1 parcel of storage for each. The variable space begins immediately after the last instruction in the program. The listing lists variables and their allocated addresses. Each symbol in the listing is identified with its value (address if a variable, CON, or label) and type. Types of indicators are V (variable), C (CON), L (label), or E (equate). Variables are allocated in the order they are defined (first referenced) in the program.

Assembly Errors

Two types of messages are issued by the assembler: errors and warnings. Errors are generally serious and inhibit code generation for the line with the error; instead of generating code, the assembler generates a PASS instruction. A warning is less serious and may still allow code to be generated. Error messages appear both on the stderr output (user's terminal by default) and in the listing file. If an error line begins with a label, then the label definition is not removed, so the label points to the line with the PASS instruction.

Instruction Reference

The following pages contain a reference table listing the interpreter's built-in instructions. Refer to the interpreter documentation for a more detailed description. Refer to the documentation for individual device drivers for a description of device-specific opcodes.

Table 1. Macrocode Built-in Instructions

Mnemonic	Opcode	Arguments
PASS	000	None
END	001	1
JUMP	002	1
LOOP	003	1, 2
IFEQ	004	1, 2, 3
IFNE	005	1, 2, 3
IFLT	006	1, 2, 3
IFGT	007	1, 2, 3
IFLE	010	1, 2, 3
IFGE	011	1, 2, 3
IFPS	012	1, 3
IFNG	013	1, 3
ANDL	014	1, 2, 3
ADDL	015	1, 2, 3
SUBL	016	1, 2, 3
EXOR	017	1, 2, 3
SHFR	020	1, 2, 3
SHFL	021	1, 2, 3
GENUSR	022	1, 2, 3
COMP	023	1, 2, 3
RAN	024	1, 2, 3
OR	025	1, 2, 3
PLANT	026	1, 2, 3
ASGN	030	1, 2
GEND	031	1, 2, 3
RPTERR	032	1, 2, 3
IPC	034	None

Table 1. Macrocode Built-in Instructions (continued)

Mnemonic	Opcode	Arguments
IEC	035	None
LIND	036	1, 2
SIND	037	1, 2
VERD	040	1, 2, 3
MOVP	041	1, 2
STOP	042	1
SETPARM	043	1, 2, 3
PUT	044	1, 2, 3
GET	045	1, 2, 3
SHFRC	046	1, 2, 3
SHFLC	047	1, 2, 3
SAVEBUF	050	1, 2
LOADBUF	051	1, 2
MUXOUT	054	1, 2
EIOX	055	1, 2
JMPSR	056	1
EXIT	057	None
WIOB	060	1, 2, 3
RIOB	061	1, 2, 3
SETPTR	062	1, 2, 3
CSTAT	063	1, 2, 3
CVAR	064	1, 2
FLOADBUF	065	1, 2, 3
FSAVEBUF	066	1, 2, 3

Reader Comment Form

Title: **DMS Macrocode Assembler Programmer Reference Manual** **Number:** **CSM-1023-000**

Your feedback on this publication will help us provide better documentation in the future. Please take a moment to answer the few questions below.

For what purpose did you primarily use this manual?

_____Troubleshooting
_____Tutorial or introduction
_____Reference information
_____Classroom use
_____Other - please explain _____

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria and explain your ratings:

_____Accuracy _____
_____Organization _____
_____Readability _____
_____Physical qualities (binding, printing, page layout) _____
_____Amount of diagrams and photos _____
_____Quality of diagrams and photos _____

Completeness (Check one)

_____Too much information _____
_____Too little information _____
_____Just the right amount of information

Your comments help Hardware Publications and Training improve the quality and usefulness of your publications. Please use the space provided below to share your comments with us. When possible, please give specific page and paragraph references. We will respond to your comments in writing within 48 hours.

NAME_____
JOB TITLE_____
FIRM_____
ADDRESS_____
CITY_____STATE_____ZIP_____
DATE_____

[or attach your business card]

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attn: Hardware Publications & Training
770 Industrial Boulevard
Chippewa Falls, WI 54729

Fold

STAPLE