M T S

The Michigan Terminal System

MTS Command Extensions and Macros

Volume 21

Reference R1021

April 1986

Updated February 1991

University of Michigan
Information Technology Division
Consulting and Support Services

DISCLAIMER

The MTS manuals are intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the *U-M Computing News* and other ITD documentation for the latest information about changes to MTS.

CONTENTS

Preface 7

Preface to Volume 21 9

Tutorial Section 11
- Introduction 13
  - Overview 13
  - On-Line EXPLAIN Facility 15
  - The Macro Flag Character 15
  - Comments 16
  - System Variables 16
  - User Variables 17
  - Expressions 18
  - Built-In Functions 20
  - The IF Command 21
  - Macros 23
  - Input and Output 27
  - Iteration 29
  - Testing and Debugging Macros 32
  - Macro Libraries 35
  - System Macro Library 37

Reference Section 39
- General Comments 41
  - I/O Stream Macro Processor Overview 41
  - The EXPLAIN Facility 42
  - Macro Flag Character 43
  - Error Messages 43
- Variables and Expressions 45
  - Variables 45
    - Defining a Variable 45
    - Types 46
    - Aggregates: Arrays and Structures 46
      - Arrays 46
      - Structures 46
      - Static Use 47
      - Dynamic Use 47
    - Named Constants 47
    - Scope 47
      - Order of Resolution 48
      - Overriding the Default 48
    - Disposing of Variables 48
    - Using Variables 48
    - Setting Values 49

Displaying Values 49
Expressions 51
Operators 51
Variables 55
Constants 56
Built-In Functions 57
ABS 57
ABSTIME 57
ARRAY 58
CAN_CONVERT 58
CONTROL 58
COST 59
CUINFO 59
DOUBLE 60
DUPL 60
EVAL 60
FILE 61
FILE_INFO 61
FILES 62
FIND_CHAR 62
FIND_STRING 63
GUINFO 63
IGNORE_CHAR 63
INTEGER_PART 64
LOCK 64
LOWERCASE 64
LPAD 64
LTRIM 65
MAX 65
MAX_SUBSCRIPT 65
MICROSECONDS 65
MIN 66
MIN_SUBSCRIPT 66
QUOTE 66
RELTIME 66
REPLACE 67
REVERSE 67
RPAD 67
SENSE 68
SIZE 68
SUBSTITUTE 68
TRIM 68
UPPERCASE 69
VARIABLE 69
System Variables 69
BATCH 69
CLS_NAME 70
CPU_TIME 70
CS_CODE 70
CS_ORIGIN 70
CS_SUMMARY 70
DATE 70

FULL_SCREEN_POSSIBLE 70
HOST_NAME 70
INSTALLATION 70
INSTALLATION_CODE 71
INSTALLATION_NAME 71
LAST_SIGNON 71
NONNULL 71
NULL 71
PROJECT 71
RATE 71
RUNRC 71
SIGNONID 71
TASKNBR 71
TERMTYPE 71
TIME 72
How Expressions Are Formed 72
Automatic Type Conversions in Expressions 73
Control Structures 73
Labels 73
Transfer of Control 74
Conditionals 74
Iteration 75
Iteration Exits 77
Comments 77
Continuation Lines 78
Command Macros 79
How to Invoke 79
Where the Definition Comes From 79
Sources 79
Search Order 80
Scope 81
Defining a Macro 81
Macro Parameters 82
Positional Parameters 82
Keyword Parameters 82
Valueless Keyword Parameters 83
Detailed Syntax of the Macro Prototype 83
Invocation of a Macro 84
Lists and Sublists 86
Detailed Syntax of the Macro Call 86
Exiting a Macro or User-Defined Function 88
Exiting a Macro 88
Exiting a User-Defined Function 88
Common to Both 88
Control and Monitoring 88
MACROECHO Option 89
MACROTRACE Option 89
The @CHECK Option 89
Input and Output From Macros 90
Input 90
Output 92
User-Defined Functions 93

Macro Libraries 95
Using a Macro Library 95
Multiple Libraries 96
Form of a Macro Library 96
Constructing a Macro Library 96
System Macro Library 96
Editing Macros 97
Special Cases 97
Emitting Lines 97
Generating Unmentionables 98
Filters 98
Multiple Definitions 99
Editing 99
Appendix A: MACRO and MACRO_FLAG_REQUIRED Modifiers 100
Appendix B: Interactions of Flag Characters and Modes 101
Appendix C: CSGET and CSSET Subroutine Descriptions 102
Appendix D: Macro Library Utilities 105
Appendix E: Examples 107
Appendix F: Macro Definition and Expansion 114
Appendix G: Input Time Conversion 118
Appendix H: Additional Built-In Functions 121
BITAND, BITNOT, BITOR 121
EXTERNAL 121
HEX 121
INTEGER_FORMAT 122
SHIFT_LEFT, SHIFT_RIGHT 122
TIME_FORMAT 122
TIME_WAIT 123
TRANSLATE 124
USE_AS 124
Appendix I: Integer Picture Formats 125
Appendix J: Time Picture Formats 127
Appendix K: Subroutines to Access Macro Variables 131
Appendix L: Symbol Index 133

Index 139

PREFACE

The software developed by the Information Technology Division (ITD) technical staff for the operation of the high-speed IBM 370-compatible computers can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The *MTS Volumes* are a series of manuals that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of ITD and the physical facilities provided are described in other publications.

The *MTS Volumes* now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are periodically updated, users should check the file *CCPUBLICATIONS, or watch for announcements in the *U-M Computing News*, to ensure that their volumes are fully up to date.

Volume 1: *The Michigan Terminal System*, Reference R1001, November 1988
Volume 2: *Public File Descriptions*, Reference R1002, January 1987
Volume 3 *System Subroutine Descriptions*, Reference R1003, April 1981
Volume 4: *Terminals and Networks in MTS*, Reference R1004, July 1988
Volume 5: *System Services*, Reference R1005, May 1983
Volume 6: *FORTRAN in MTS*, Reference R1006, October 1983
Volume 7: *PL/I in MTS*, Reference R1007, September 1982
Volume 8: *LISP and SLIP in MTS*, Reference R1008, June 1976
Volume 9: *SNOBOL4 in MTS*, Reference R1009, September 1975
Volume 10: *BASIC in MTS*, Reference R1010, December 1980
Volume 11: *Plot Description System*, Reference R1011, August 1978
Volume 12: *PIL/2 in MTS*, Reference R1012, December 1974
Volume 13: *The Symbolic Debugging System*, Reference R1013, September 1985
Volume 14: *360/370 Assemblers in MTS*, Reference R1014, May 1983
Volume 15: *FORMAT and TEXT360*, Reference R1015, April 1977
Volume 16: *ALGOL W in MTS*, Reference R1016, September 1980
Volume 17: *Integrated Graphics System*, Reference R1017, December 1980
Volume 18: *The MTS File Editor*, Reference R1018, February 1988
Volume 19: *Tapes and Floppy Disks*, Reference R1019, October 1989
Volume 20: *Pascal in MTS*, Reference R1020, December 1985
Volume 21: *MTS Command Extensions and Macros*, Reference R1021, April 1986
Volume 22: *Utilisp in MTS*, Reference R1022, May 1988
Volume 23: *Messaging and Conferencing in MTS*, Reference R1023, February 1991

The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. *MTS Volume 1*, for example, introduces the user to MTS and describes in general the MTS operating system, while *MTS Volume 10* deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury
General Editor

PREFACE TO VOLUME 21

MTS Volume 21: MTS Command Extensions and Macros, combines Computing Center Memo 464, *Introduction to MTS Command Extensions and Macros*, and the document *Command Extensions and Macros*, which was written by Donald W. Boettner of the University of Michigan Information Technology Division.

This volume is divided into two sections, a tutorial section and a reference section. The tutorial section presumes only a basic knowledge of the MTS command language. However, the material presented in the reference section presumes that the reader has a fundamental knowledge of computer programming languages.

TUTORIAL SECTION

INTRODUCTION

OVERVIEW

The MTS command extension and macro processor is a special set of system programs that act as an interface between the user and MTS. The processor has the following capabilities:

(1) Provides a series of command extensions which allow the setting and displaying of system variables that describe the current status of MTS.

(2) Provides conditional commands that allow flow control depending on the results of previous commands, programs, or system variables.

(3) Allows the user to create procedures (or macros).

The macro processor is enabled by issuing the MTS command

```
$SET MACROS=ON
```

Once the macro processor is enabled, the command extensions, which are accessed by macro commands, may be used. The remainder of this description assumes that the macro processor has been enabled unless otherwise indicated.

When the macro processor is enabled, it looks at all lines that are read from the terminal (*SOURCE*). Lines that are macro commands, identified by a ">" character in column 1, are processed by the macro processor. Lines that are not macro commands are passed onto the system for normal processing (this includes MTS commands, file editor commands, program data lines, etc.).

Macro commands can be used in two ways:

(1) they can be executed immediately, or

(2) they can be stored in macro definitions, where execution is deferred until the macro is invoked.

Except when defining a macro, macro commands are executed immediately when entered from a terminal or read from a $SOURCE file (or a sigfile, a special type of source file that is processed at signon time). For example, entering the macro command

```
>DISPLAY SYSTEM_VARIABLES
```

displays the list of predefined system variables available with the macro processor, while entering the macro commands

```
>DISPLAY VALUE(SIGNONID)
>DISPLAY VALUE(PROJECT)
```

displays the current contents of the system variables SIGNONID (user signon ID) and PROJNO (user project ID). The following example illustrates how a series of macro commands may be used to customize a shared sigfile for a specific user:

```
$SET MACROS=ON
>*
>* Execute for all users
>*
$MESS RETRIEVE NEW HEADER ALL
$SET AUTOHOLD=ON
$SET INITFILE(EDIT)=WABC:EDIT.INITF
>*
>* Execute only for Ontel terminals
>*
>IF TERMTYPE='OP/VIS    ', $CONTROL *MSOURCE* PAD=PFKEYS
>*
>* Set the user's name for $MESSAGE system
>*
>IF SIGNONID='WABC'
     $SET NAME='Marlou Smith'
     $SET ROUTE=UNYN
>ELSEIF SIGNONID='WXYZ'
     $SET NAME='John Brown'
     $SET ROUTE=NUBS
>ENDIF
```

For further information on the use of sigfiles, see MTS Volume 1, *The Michigan Terminal System*.

A macro is a user-defined command or procedure. A macro has a name which is associated with a single command or a sequence of commands that make up its definition. Once the macro is defined, it can be subsequently executed by simply entering the macro name with any necessary parameters. For example, a simple one-command macro may be defined as follows:

```
>MACRO PAGEPR file
$RUN *PAGEPR SCARDS={file} PAR=PORTRAIT,PAPER=PLAIN
>ENDMACRO
```

This defines a macro named PAGEPR to run the *PAGEPR program. It has one parameter called "file" which specifies the file to be printed. The PAGEPR macro may be executed by giving the command

```
>PAGEPR DOCUMENT
```

This is equivalent to issuing the MTS command

```
$RUN *PAGEPR SCARDS=DOCUMENT PAR=PORTRAIT,PAPER=PLAIN
```

The following example defines a macro to be used for compiling and executing a Pascal/VS program:

```
>MACRO PASCAL source object
$RUN *PASCALVS SCARDS={source} SPUNCH={object}
>IF RUNRC > 0, EXIT
$RUN {object}
>ENDMACRO
```

The macro may be executed by giving the command

```
>PASCAL PROB1SOU PROB1OBJ
```

The macro first compiles the Pascal source code in the file PROB1SOU. If the compilation is successful, the object code placed in the file PROB1OBJ is then executed; otherwise, the macro is exited

(stops) without running the object program. This example illustrates the power that macros have for making decisions based on the results of previous commands.

Macros that will be used for most or all sessions can be defined directly in the user's sigfile. The macro processor also provides a more sophisticated facility for creating and maintaining macro libraries which allows the user to selectively define macros for each session depending upon what tasks are to be performed within the session. This facility is explained later.

ON-LINE EXPLAIN FACILITY

The macro processor has an on-line EXPLAIN facility which is activated by issuing the command

>EXPLAIN item

where "item" specifies the subject to be explained. By entering

>EXPLAIN

the first time through, the user will be given a brief tutorial on the use of the EXPLAIN facility. By entering

>EXPLAIN CONTENTS

the user will be given a menu of items for which explanations are available. Once the user has invoked the EXPLAIN facility, subsequent explanations are produced by entering only the item to be explained in response to the prompt

```
Explain?
```

For example by responding with DISPLAY, an explanation of the DISPLAY macro command will be produced. The EXPLAIN facility is terminated by entering STOP. An end-of-file will also terminate the EXPLAIN facility.

THE MACRO FLAG CHARACTER

The macro flag character ">" indicates that an input line is a macro command or a macro invocation. Normally, all input lines read from *SOURCE* are inspected for macro commands and invocations. Lines read from *SOURCE* include lines entered at the user's terminal, cards read in a batch job, and lines read from a file that has been $SOURCEd.

If a line contains a macro flag character ">" in column 1, it is treated as a macro command or macro invocation. If the line is a valid macro command, it is processed by the macro processor; if the line is an invalid macro command, an error message is generated. For example, the macro command

>DISPLAY MACROS

will display the names of all user-defined macros. There are two common cases in which the ">" is not recognized as a macro flag character—lines read by the MTS file editor in insertion mode and lines read by the message system as message text are never treated as macro commands regardless of the presence of the ">" in column 1.

The flag character ">" is optional for lines within a macro definition (however, the flag character is required for the MACRO and ENDMACRO commands that delimit the definition). When the macro is invoked (executed), all lines comprising the definition are normally inspected for macro commands and invocations regardless of the presence of the flag character. If the line contains a valid macro command, it is processed as such; otherwise, it is passed on for normal processing. For example, the PASCAL macro illustrated above could have been defined as

```
>MACRO PASCAL source object
$RUN *PASCALVS SCARDS={source} SPUNCH={object}
IF RUNRC > 0, EXIT
$RUN {object}
>ENDMACRO
```

The macro flag character is also optional for macro invocation lines. For example, the above PASCAL macro could be invoked as

```
PASCAL PROB1SOU PROB1OBJ
```

Throughout the remainder of this description, all macro invocations will be given without the macro flag character.

COMMENTS

Comment lines can be used to document the purpose of macro command lines. A comment line is indicated by placing the characters ">*" in columns 1 and 2. For example, the above PAGEPR macro could have been defined with comments as follows:

```
>MACRO pagepr file
>*
>* Run *PAGEPR program in portrait mode
>*
$RUN *PAGEPR SCARDS={file} PAR=PORTRAIT,PAPER=PLAIN
>ENDMACRO
```

The three comment lines are used only for internal documentation of the macro definition are not printed by the macro processor during macro invocation.

SYSTEM VARIABLES

A variable is used to store information and has both a name and a value. The value may remain the same or may change (vary) during the course of the session. There are two types of variables that can be used with the macro processor, *system variables* and *user variables*. System variables are described below while user variables are described later.

System variables are used to store information that describes the status of the system. System variables are predefined, that is, they are created by the macro processor. Their values are automatically set by the system. Some of the more useful system variables are

BATCH	is TRUE for a batch job and FALSE for a terminal job.
CS_CODE	is the termination code set by the last command or program (zero is normal return, nonzero is error return).
DATE	is the current date (day mon dd/yy)

PROJECT	is the current project number.
RUNRC	is the return code from the last $RUN command.
SIGNONID	is the current signon ID (in uppercase).
TERMTYPE	is the 8-character terminal type.
TIME	is the current time (hh:mm.ss zon).

The value of a system variable can be displayed by issuing the command

>DISPLAY VALUE(name)

where "name" specifies the name of variable to be displayed. For example, issuing the command

>DISPLAY VALUE(DATE)

will display the date. The complete list of system variables can be obtained by issuing the command

>DISPLAY SYSTEM_VARIABLES

USER VARIABLES

User variables are defined (created) by issuing the macro command

>DEFINE name

where "name" specifies the name of the variable being defined. The name can be from 1 to 255 characters in length; it can contain only alphanumeric characters (A–Z, 0–9, plus the underscore "_"), and the first character must be alphabetic. Lowercase letters in variable names are automatically converted to uppercase during processing. Some legal user variable names are:

```
ALPHA
Beta     (same as BETA)
VAR_1
```

A value can be initially assigned to the variable by issuing the DEFINE command in the form

>DEFINE name=value

For example, the following command will define the variable N and assign it an initial integer value of 1:

```
>DEFINE N=1
```

The data type of the variable is automatically set to match the data type of the value assigned to it each time its value changes. Thus, a single variable can be initially assigned to a value of any data type and later reassigned to a value of any other data type. The most common data types are:

Type	Example Values
Integer	1, 25, 0, –5, 15000
String	"Seven", 'ABCDEF', "" (null string)
Boolean	True, False
Hexadecimal	X'6D', X'00FFFFFF'

Line Number (Real) 1.5, .333, –25.75

Boolean variables can only have the values TRUE and FALSE; they are often used in conditional expressions within the IF command (described later). Line-number variables are real values that take the form of an MTS line number—up to three positions to the right of the decimal point are allowed.

The SET macro command can be used to assign a value to an existing variable. The command is given in the form

>SET VAR name=value

For example, the variable N defined and initialized above may be reassigned to the integer value 2 by giving the command

```
>SET VAR N=2
```

The variable N could be assigned to a character string value by giving the command

```
>SET VAR N='Two'
```

The current value of N can be displayed by giving the command

```
>DISPLAY VALUE(N)
```

The complete list of currently defined user variables can be obtained by issuing the command

>DISPLAY USER_VARIABLES

By default, the scope of user variables defined within macros is *local*, that is, they can be used only within the context of the macro and are automatically destroyed when execution of the macro is terminated. The scope of all other user variables is *global*, that is, they can be used at any time either within or outside of a macro and they continue to exist until explicitly destroyed by the user. A global user variable can be destroyed (forgotten) by issuing the command

>FORGET name

Other types of user variables can be defined. These include arrays, structures, and named constants. These variable types are described in the reference documentation.

Several examples illustrating the use of user variables are given later in the discussion of macros.

EXPRESSIONS

An expression is a formula indicating how a value is to be computed. There are several types of expressions that may be used with the macro processor. This section describes *arithmetic expressions*, *string expressions*, and *logical expressions*. Other types of expressions are available and are discussed in the reference documentation.

Arithmetic expressions are used to generate numeric values. The following *arithmetic operators* may be used:

– (unary)	negation
+	addition
– (binary)	subtraction
*	multiplication
/	division

The rules for forming arithmetic expressions are same as those used in most programming languages. Parentheses are used to indicate the precedence of operators. Examples using these operators are

```
-5
I + 1
2*N - 6
(A+B+C)/10
```

System variables, user variables, and constants all can be used in expressions. The presence of blanks surrounding the operators is optional.

Arithmetic expressions can commonly used with the SET command to assign values to variables. For example, the value of N can be incremented by 1 by giving the command

```
>SET VAR N=N+1
```

String expressions are used to perform string operations. The following *string operators* may be used:

||	concatenation
string(beg|len)	substring
string(beg...end)	substring

The concatenation operator joins (concatenates) two strings to form a new and longer string. The substring operators return the portion of a string specified by either the beginning index and the length or the beginning and ending indices. Examples of string expressions are

```
"ABCDEF" || "GHI"          (returns "ABCDEFGHI")
ALPHA || BETA
"ABCDEF"(1|4)              (returns "ABCD")
"ABCDEF"(1...4)            (returns "ABCD")
ALPHA(2...6)
```

Logical expressions (or Boolean expressions) are expressions that evaluate to either true or false using the common rules of logic. The simplest logical expressions are those consisting of a single variable that has a Boolean data type. However, most logical expressions make use of one of the following *relational operators*:

=	equal to
¬=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Examples of logical expressions are

```
BATCH
```

```
SIGNONID='WABC'
CS_CODE ¬= 0
RUNRC >= 8
Var1 + 1 = Var2
ALPHA || BETA = "ABCDEF"
```

Arithmetic and string operators may be used within logical expressions.

Two special relational operators are provided for use with the FILE and VARIABLE built-in functions:

EXISTS	is nonnull
DOESNT EXIST	is null

The following *logical operators* can also be used to create more complex expressions:

NOT	logical not
AND	logical and
OR	logical or

Examples of expressions using these operators are

```
NOT BATCH
RUNRC=8 AND RUNRC=12
I = 1 OR J = 1
(I=1 OR J=1) AND 2*K=0
```

BUILT-IN FUNCTIONS

Several functions are provided to return information to the user or to perform certain operations. These functions are built into the macro processor, that is, they are predefined and always are available for use. Some of the more useful built-in functions are

UPPERCASE(string)	returns the supplied string in uppercase, e.g., UPPERCASE("abc") or UPPERCASE(INPUT).
LOWERCASE(string)	returns the supplied string in lowercase, e.g., LOWERCASE("ABC") or LOWERCASE(INPUT).
INTEGER_PART(nbr)	returns the integer portion of the number supplied, e.g., INTEGER_PART(2.5) or INTEGER_PART(COUNT/5).
SIZE(string)	returns the length of the string, e.g., SIZE("ABC") or SIZE(INPUT).
VARIABLE(string)	takes a string as an argument and returns the null string if there is no user variable of that name or returns the variable data type otherwise, e.g., VARIABLE("ALPHA").
FILE(string)	takes a string as an argument and returns the null string if there is no file of that name or returns the file name otherwise, e.g., FILE("DATA1").
FILES(string)	takes a FILESTATUS pattern as an argument and returns a list of files that match the pattern, e.g., FILES("?") or FILES("?.S").
GUINFO(item)	takes an GUINFO item name or numerical index and returns the associated information, e.g., GUINFO("RCPRINT") or GUINFO(23). See MTS Volume 3, *System Subroutine Descriptions*, for a list of the GUINFO items available.

Each of the above functions takes an argument and returns a value. These functions are normally used in expressions. For example, the following command

```
>IF UPPERCASE(INPUT) = "YES", $SIGNOFF
```

will execute the $SIGNOFF command if the uppercase conversion of input is "YES", that is, "YES", "yes", and "Yes" all will match "YES". The following command

```
>IF N/3=INTEGER_PART(N/3), ...
```

can be used to test whether the value of N is a multiple of 3 (e.g., if N=6, the expression reduces to 2=2 which is TRUE, but if N=7, the expression reduces to 2.333=2 which is FALSE). The IF command is described in more detail in the next section.

Other examples of using built-in functions are given in the remaining text. A complete list of all of the available built-in functions is given in the reference documentation.

THE IF COMMAND

The IF macro command allows the user to conditionally execute a command or a sequence of commands. The most elementary form of the IF command is

```
>IF bexp, statement
```

where "bexp" is a Boolean expression that evaluates to either TRUE or FALSE and "statement" is a single command to be executed if the expression is TRUE. For example, the command

```
>IF SIGNONID='WABC', $SET NAME='Marlou Smith'
```

will set the user's name to Marlou Smith if the current signon ID is WABC.

A second form of the IF command which executes a sequence of commands is

```
>IF bexp
    statement
      ...          These executed if "bexp" is TRUE
      ...
    statement
>ENDIF
```

In this case if the expression "bexp" is TRUE, the set of specified statements is executed. For example, the command

```
>IF SIGNONID='WABC'
   $SET NAME='Marlou Smith'
   $SET ROUTE=UNYN
>ENDIF
```

will set the user's name to Marlou Smith and route all output to the UNYN batch station if the signon ID is WABC.

A variation of this form of the IF command, which executes an alternative sequence of commands if the expression is FALSE, is

```
>IF bexp
    statement
      ...          These executed if "bexp" is TRUE
      ...
    statement
>ELSE
    statement
      ...          These executed if "bexp" is FALSE
      ...
    statement
>ENDIF
```

In this case if the expression "bexp" is TRUE, the first set of statements are executed; otherwise, the second set of statements are executed. For example, the command

```
>IF SIGNONID='WABC'
   $SET NAME='Marlou Smith'
   $SET ROUTE=UNYN
>ELSE
   $SET ROUTE=CNTR
>ENDIF
```

will set the user's name to Marlou Smith and route all output to the UNYN remote station if the signon ID is WABC; otherwise (when the signon ID is not WABC and therefore the expression is FALSE), all output will be routed to the Computing Center station.

The complete form of the IF command, which allows the testing of several expressions, is

```
>IF bexp1
    statement
      ...          These executed if "bexp1" is TRUE
      ...
    statement
>ELSEIF bexp2
    statement
      ...          These executed if "bexp1" is FALSE
      ...          and "bexp2" is TRUE
    statement
>ELSE
    statement
      ...          These executed if "bexp1" is FALSE
      ...          and "bexp2" is FALSE
    statement
>ENDIF
```

In this case if the expression "bexp1" is TRUE, the first set of statements are executed; otherwise, if the expression "bexp1" is FALSE and the expression "bexp2" is TRUE, the second set of statements are executed; otherwise (when both "bexp1" and "bexp2" are FALSE), the last set of statements are executed (note that only one set of statements is executed). For example, the command

```
>IF SIGNONID='WABC'
   $SET NAME='Marlou Smith'
   $SET ROUTE=UNYN
>ELSEIF SIGNONID='WXYZ'
   $SET NAME='John Brown'
   $SET ROUTE=NUBS
>ELSE
   $SET ROUTE=CNTR
>ENDIF
```

will set the user's name to Marlou Smith and route all output to the UNYN remote station if the signon ID is WABC; or, if the signon ID is WXYZ, will set the user's name to John Brown and route all output to the NUBS remote station; otherwise (when the signon ID is neither WABC nor WXYZ and therefore both expressions are FALSE), all output will be routed to the Computing Center station.

Additional ELSEIF sections can be inserted into the IF command structure to test other conditions.

MACROS

A macro is a user-defined command that consists of a series of commands or steps. The power of a macro lies in its ability to combine a sequence of commands into a single procedure that can be executed by a single command.

Before a macro can be used, it must be defined by the user. The most elementary form of a macro definition is

```
>MACRO macro-name
    statement
      ...
      ...
    statement
>ENDMACRO
```

where "macro-name" is the name of the macro being defined and the "statements" are lines to be produced when the macro is invoked. The lines can be macro commands which are used to control the execution of the macro, or they can be program data lines, MTS commands, or other types of commands (e.g., $EDIT commands or $MESSAGE commands) which are *emitted* by the macro processor for processing by programs or other command processors. The rules for forming macro names are the same as for defining variable names.

The following commands define a macro to run the *PAGEPR program.

```
>MACRO pagepr
$RUN *PAGEPR SCARDS=DOCUMENT PAR=PORTRAIT,PAPER=PLAIN
$RELEASE *PRINT*
>ENDMACRO
```

The file DOCUMENT will contain the output to be printed using the PORTRAIT font with PLAIN (unpunched) paper. After the file is printed, *PRINT* will be automatically released. The macro may be executed by issuing the command

```
PAGEPR
```

which is equivalent to issuing the MTS commands

```
$RUN *PAGEPR SCARDS=DOCUMENT PAR=PORTRAIT,PAPER=PLAIN
$RELEASE *PRINT*
```

One major difficulty with a macro defined in this manner is that it is not very flexible. That is, a separate macro must be defined for each different output file that is to be printed. Also, if other fonts or other types of paper stock are desired, separate macros will have to be defined.

These limitations can be overcome by the use of *positional parameters*, which allow the user to specify variations that can be made to the commands executed when the macro is invoked. Positional parameters are declared in the MACRO prototype statement by appending them after the macro name, i.e.,

>MACRO macro-name par1 par2 ...

where "pars" specify the positional parameters. For example, a more flexible definition of the PAGEPR macro using three positional parameters is

```
>MACRO pagepr file font stock
$RUN *PAGEPR SCARDS={file} PAR={font},PAPER={stock}
$RELEASE *PRINT*
>ENDMACRO
```

In this case, the positional parameters "file", "font", and "stock" specify variations to the $RUN *PAGEPR command. In the $RUN *PAGEPR line of the above definition, the three parameters must be enclosed in braces to indicate that values are to be substituted for them during execution. A further discussion about parameter substitution and the use of braces is given below.

When the macro is invoked, the values to be substituted into the definition are given on the invoking command (in the same position (order) as they are declared in macro prototype statement). For example, the macro invocation

```
PAGEPR DOCUMENT PORTRAIT PLAIN
```

executes the $RUN *PAGEPR command as

```
$RUN *PAGEPR SCARDS=DOCUMENT PAR=PORTRAIT,PAPER=PLAIN
```

while the macro invocation

```
PAGEPR DOCUMENT LANDSCAPE 3HOLE
```

executes the $RUN *PAGEPR command as

```
$RUN *PAGEPR SCARDS=DOCUMENT PAR=LANDSCAPE,PAPER=3HOLE
```

If a positional parameter is omitted during invocation, the macro processor will prompt for it.

Keyword parameters also can be used to specify variations to the manner in which a macro is executed. Keyword parameters are declared in the MACRO prototype statement, i.e.,

>MACRO macro-name keyname1=value keyname2=value ...

where “keyname” is the name of the keyword parameter and “value” is its default value if it is not specified on invocation. Both positional and keyword parameters can be used in the same macro definition. For example, the PAGEPR macro could be defined to use one positional parameter and two keyword parameters:

```
>MACRO pagepr file font=portrait stock=plain
$RUN *PAGEPR SCARDS={file} PAR={font},PAPER={stock}
$RELEASE *PRINT*
>ENDMACRO
```

In this case, the macro invocation

```
PAGEPR DOCUMENT
```

executes the $RUN *PAGEPR command as

```
$RUN *PAGEPR SCARDS=DOCUMENT PAR=PORTRAIT,PAPER=PLAIN
```

using the defaults of PORTRAIT and PLAIN specified in the macro prototype statement, while the macro invocation

```
PAGEPR DOCUMENT FONT=LANDSCAPE STOCK=3HOLE
```

executes the $RUN *PAGEPR command as

```
$RUN *PAGEPR SCARDS=DOCUMENT PAR=LANDSCAPE,PAPER=3HOLE
```

Keyword parameters can be given in any order on invocation.

Macro commands as well MTS commands can be used in the definition of a macro. A variation of the earlier PASCAL macro definition allows the user to specify the error-return level from *PASCALVS above which object program execution will be suppressed.

```
>MACRO PASCAL source object rc
$RUN *PASCALVS SCARDS={source} SPUNCH={object}
IF RUNRC > rc, EXIT
$RUN {object}
>ENDMACRO
```

For example, the macro invocation

```
PASCAL SOU OBJ 0
```

will terminate macro processing whenever a compilation return code greater than 0 is generated by the *PASCALVS compiler, while

```
PASCAL SOU OBJ 4
```

will terminate macro processing whenever the return code from *PASCALVS is greater than 4 (that is, program execution will not be suppressed if only compiler warning messages are generated).

Normally, the statements of a macro are executed in sequence until the last statement has been executed. The EXIT statement can be used to halt execution at an earlier time as illustrated in the above example.

The above example further illustrates the use of braces in a macro definition. In general, braces are not required whenever a parameter is used within a macro command in the definition (in this case, the >IF command). However, the braces are required for all other types of commands that are emitted, that is not directly processed by the macro processor but sent on to the system for processing.

Macros also can contain commands other than MTS commands or macro commands. The following example illustrates how a small file editor macro can be defined to shift a file an arbitrary number of column positions to the left or right.

```
>MACRO SHIFT file direction count
$EDIT {file}
SHIFT /F {direction} {count} @NV
STOP
>ENDMACRO
```

In this case, the macro processor emits to the system the three commands comprising the definition. The $EDIT command is processed by MTS and the SHIFT and STOP commands are processed by the file editor. The file DOCUMENT could be shifted to the right by 2 columns by giving the command

```
SHIFT DOCUMENT RIGHT 2
```

The following macro could be defined to retrieve all new messages from the message system:

```
>MACRO MSGS
$MESSAGE
RETRIEVE NEW ALL
STOP
>ENDMACRO
```

Like the previous example, one command is processed by MTS and two commands are processed by another command processor, in this case the message system. This example could be enhanced to allow the user to route the retrieved messages to a file instead of printing them at the terminal by including the message command SET OUTPUT=FDname in the definition. However, a potential conflict arises at this point since the SET command is also a valid macro command and would be executed as such by the macro processor when the macro was invoked. To avoid this conflict, the EMIT command can be used to force the command to be emitted to the system instead of being executed by the macro processor. The EMIT command has the form

EMIT "statement"

The MSGS macro thus could be defined as follows:

```
>MACRO MSGS OUTPUT=*SINK*
$MESSAGE
EMIT "SET OUTPUT={output}"
RETRIEVE NEW ALL
STOP
>ENDMACRO
```

If the macro were invoked by the command

```
MSGS
```

the retrieved messages would be printed at the terminal, but if the macro were invoked by the command

```
MSGS OUTPUT=*PRINT*
```

the messages would be printed on *PRINT*.

INPUT AND OUTPUT

The macro commands READ and WRITE can be used to perform input and output. They can be executed immediately or can be used in macro definitions.

The READ command is used to read an input line and is given in the form

READ variable [FROM FDname]

where "variable" specifies a user variable that the input line is to be read into and "FROM FDname" specifies where the input is being read from. If the FROM clause is omitted, the line is read from the *SOURCE* (normally the user's terminal).

The WRITE command is used to write an output line and is given in the form

WRITE expression [ON FDname]

where "expression" specifies an expression to be written and "ON FDname" specifies where the output line is to be written. If the ON clause is omitted, the expression is written to *SINK* (normally the user's terminal).

The following example is a variation of the above PASCAL macro which prompts the user whether or not to execute the object program when there is an error during compilation:

```
>MACRO PASCAL source object
$RUN *PASCALVS SCARDS={source} SPUNCH={object}
IF RUNRC > 0
  DEFINE ANSWER
  WRITE "Compilation error"
  WRITE "Enter OK to continue"
  READ ANSWER
  IF UPPERCASE(ANSWER)¬="OK", EXIT
ENDIF
$RUN {object}
>ENDMACRO
```

In this example, the variable ANSWER is used to read the user's response to the prompting message. Since this is a user variable, it must be defined by the DEFINE command before it is used. The response to the prompt will be read from *SOURCE* since the FROM FDname clause is omitted from the READ command. The built-in function UPPERCASE returns the value of ANSWER in uppercase letters so that all case forms of "OK" (i.e., OK, Ok, oK, ok) can be checked for in the prompt response.

The following IF-command segment illustrates how the sigfile example presented at the beginning of this description could be enhanced with READ and WRITE commands:

```
>IF SIGNONID='WABC'
   $SET NAME='Marlou Smith'
   $SET ROUTE=UNYN
>ELSEIF SIGNONID='WXYZ'
   $SET NAME='John Brown'
```

```
   $SET ROUTE=NUBS
>ELSE
>  DEFINE NAME
>  DEFINE SITE
>  WRITE 'Enter name:'
>  READ NAME FROM *MSOURCE*
>  $SET NAME='{NAME}'
>  WRITE 'Enter output site:'
>  READ SITE FROM *MSOURCE*
>  $SET ROUTE={SITE}
>ENDIF
```

In this example, two points must be noted. First, the FROM *MSOURCE* clause must be specified on both READ commands so that the input lines are read from the user's terminal (*MSOURCE* always refers to the user's terminal regardless of the setting of *SOURCE*). If FROM *MSOURCE* were not specified, the input lines would be read from *SOURCE* (the default) which in this case would be the next line of the sigfile (remember that a sigfile is a special case of a $SOURCE file and therefore *SOURCE* refers to the lines of the sigfile). The second point is that the macro flag character must be used to denote all macro commands in the ELSE clause since these commands are not a part of a macro definition (as opposed to the PASCAL macro definition above).

When the ELSE clause is executed in the above example, the user will be prompted in the form

```
 Enter name:
?
 Enter output site:
?
```

with the responses being entered on the lines following the prompting messages. A more elegant form of prompting can be produced by using the PREFIX clause on the READ command, which is given in the form

READ variable [FROM FDname] [PREFIX=string]

where PREFIX=string specifies a prompting prefix to be used when reading the line. Normally, this prefix is a character string and must be enclosed in quotes or primes. The input sequence of the above example could be rewritten as

```
   ...
>ELSE
>  DEFINE NAME
>  DEFINE SITE
>  READ NAME FROM *MSOURCE* PREFIX="Enter name:"
>  $SET NAME='{NAME}'
>  READ SITE FROM *MSOURCE* PREFIX="Enter output site:"
>  $SET ROUTE={SITE}
>ENDIF
```

In this case, the user will be prompted with

```
Enter name:
Enter output site:
```

The responses are entered on the same line as the prompting messages.

Two other clauses to control end-of-file and attention-interrupt processing can be specified with the READ command. They are given in the following form

READ variable [FROM FDname] [EOF–>var1] [ATTN–>var2]

where "var1" and "var2" specify user variables that are set to the value TRUE whenever an end-of-file or an attention interrupt occurs during the read operation and FALSE otherwise. For example, the first READ command of the above example could be altered so that the $SET NAME and $SET ROUTE commands are skipped if the user responds to the name prompt with an end-of-file:

```
   ...
>ELSE
>  DEFINE NAME
>  DEFINE SITE
>  DEFINE ENDF
>  READ NAME FROM *MSOURCE* PREFIX="Enter name:" EOF->ENDF
>  IF NOT ENDF, EXIT
>  $SET NAME='{NAME}'
>  READ SITE FROM *MSOURCE* PREFIX="Enter output site:"
>  $SET ROUTE={SITE}
>ENDIF
```

ITERATION

The LOOP command can be used to perform iteration within a macro. The basic form of the LOOP command is

LOOP [how]
statement
...
...
statement
ENDLOOP

The statements in the loop are executed repeatedly until the loop is terminated, either by an EXITLOOP statement or by satisfying the conditions specified by the "how" clause. For example, the following macro may be defined to compact a series of files.

```
>MACRO COMPACT
DEFINE FILENAME
DEFINE ENDF
IF FILE("TEMP") EXISTS
  WRITE "File TEMP already exists"
  EXIT
ENDIF
LOOP
  READ FILENAME EOF->ENDF
  IF ENDF, EXITLOOP
  $DUPLICATE {FILENAME} AS TEMP
  IF CS_CODE¬=0
    WRITE "Duplication of {FILENAME} unsuccessful"
    EXIT
  ENDIF
  $DESTROY {FILENAME} OK
  $RENAME TEMP AS {FILENAME} OK
  IF CS_CODE¬=0
```

```
    WRITE "Renaming of {FILENAME} unsuccessful"
    EXIT
  ENDIF
ENDLOOP
>ENDMACRO
```

In this example, the user will be prompted repeatedly to enter file names for compaction until an end-of-file is entered at which point the variable ENDF becomes TRUE and the EXITLOOP statement is executed to terminate the looping. A slightly more sophisticated example of a file compaction macro is given below. In this example, the $FILESTATUS command is executed to list all of the user's files into a temporary file –X. When the loop is then executed, the individual file names are read from –X and compacted. An error exit is included to exit the macro if an error occurs during the compaction operations.

```
>MACRO COMPACTALL
DEFINE FILENAME
DEFINE ENDF
IF FILE("TEMP") EXISTS
  WRITE "File TEMP already exists"
  EXIT
ENDIF
$EMPTY -X
$FILESTATUS ? OUTPUT=-X
LOOP
  READ FILENAME FROM -X EOF->ENDF
  IF ENDF, EXITLOOP
  $DUPLICATE {FILENAME} AS TEMP
  IF CS_CODE¬=0
    WRITE "Duplication of {FILENAME} unsuccessful"
    EXIT
  ENDIF
  $DESTROY {FILENAME} OK
  $RENAME TEMP AS {FILENAME} OK
  IF CS_CODE¬=0
    WRITE "Renaming of {FILENAME} unsuccessful"
    EXIT
  ENDIF
ENDLOOP
>ENDMACRO
```

The "how" clause can be specified in several ways. A simple loop counter can be specified by giving the form

LOOP FROM beg-exp TO end-exp

With this form, an internal variable (counter) is established to control the number of iterations of the loop. The counter is incremented starting with the value specified by the beginning expression "beg-exp" up to the value specified by the ending expression "end-exp" (by default, the counter is incremented by 1). For example, a version of the PAGEPR macro may be defined to print a specified number of copies of output:

```
>MACRO PAGEPR FILE COPIES=1
LOOP FROM 1 TO COPIES
  $RUN *PAGEPR SCARDS={FILE} PAR=PORTRAIT,PAPER=PLAIN
ENDLOOP
$RELEASE *PRINT*
>ENDMACRO
```

In this case, the value specified for the COPIES keyword parameter on the macro invocation is substituted into the TO clause of the LOOP command to set the upper limit for the number of copies wanted. If COPIES is omitted on the invocation, the count is set to one and the loop is executed once.

One limitation of this form of the FROM-TO clause is that the value of the loop counter cannot be accessed within the statements of the loop, for example, tests cannot be made for intermediate values. This limitation can be overcome by using the FOR clause in the form

LOOP FOR var FROM beg-exp TO end-exp

where "var" specifies a user variable that is used for the counter. For example, the above PAGEPR macro could be rewritten to release *PRINT* after every 10 jobs is printed by *PAGEPR:

```
>MACRO PAGEPR FILE COPIES=1
DEFINE CNTR
LOOP FOR CNTR FROM 1 TO COPIES
  $RUN *PAGEPR SCARDS={FILE} PAR=PORTRAIT,PAPER=PLAIN
  IF CNTR/10=INTEGER_PART(CNTR/10), $RELEASE *PRINT*
ENDLOOP
$RELEASE *PRINT*
>ENDMACRO
```

INTEGER_PART(exp) is a built-in function that returns the integer portion of its argument. The left-hand side of the IF-command expression CNTR/10 will be integral whenever the value of CNTR is a multiple of 10 and thus equal to the result returned by INTEGER_PART(CNTR/10).

Instead of incrementing a counter, a list of elements can also be used to control the looping. The form

LOOP FOR var OVER list

specifies a list of elements, each of which will be assigned to the variable "var" during iteration. The first element of the list will be assigned to "var" during the first iteration, the second element will be assigned to "var" during the second iteration, etc. The number of elements in the list specifies the total number of iterations of the loop.

The above COMPACTALL macro example can be rewritten to take advantage of the OVER clause. The FILES("?") built-in function returns a list of all the user's files. The OVER clause will assign to the variable FILENAME each of the file names in the list in succession as it iterates over the list.

```
>MACRO COMPACTALL
DEFINE FILENAME
IF FILE("TEMP") EXISTS
  WRITE "File TEMP already exists"
  EXIT
ENDIF
LOOP FOR FILENAME OVER FILES("?")
  $DUPLICATE {FILENAME} AS TEMP
  IF CS_CODE¬=0
    WRITE "Duplication of {FILENAME} unsuccessful"
    EXIT
  ENDIF
  $DESTROY {FILENAME} OK
  $RENAME TEMP AS {FILENAME} OK
  IF CS_CODE¬=0
    WRITE "Renaming of {FILENAME} unsuccessful"
```

```
    EXIT
  ENDIF
ENDLOOP
>ENDMACRO
```

Two other forms of the "how" clause can be used to control looping while or until a specified condition is satisfied. These forms are

LOOP WHILE bexp

which continues the loop while the expression "bexp" remains TRUE, and

LOOP UNTIL bexp

which continues the loop until the expression "bexp" becomes TRUE. In both of these forms (as well in all the above forms), the expression is evaluated only at the top of the loop (the LOOP statement), which means that if the condition becomes satisfied during the execution of the statements within the loop, the looping will not terminate until beginning of the next iteration. In general, the WHILE and UNTIL forms are used whenever the looping condition becomes TRUE or FALSE as a result of the statements executed within the loop. The previous COMPACT macro could be rewritten (albeit less efficiently) to use the UNTIL clause as follows:

```
>MACRO COMPACT
DEFINE FILENAME
DEFINE ENDF=FALSE
IF FILE("TEMP") EXISTS
  WRITE "File TEMP already exists"
  EXIT
ENDIF
READ FILENAME EOF->ENDF
IF ENDF, EXIT
LOOP UNTIL ENDF
  $DUPLICATE {FILENAME} AS TEMP
  IF CS_CODE¬=0
    WRITE "Duplication of {FILENAME} unsuccessful"
    EXIT
  ENDIF
  $DESTROY {FILENAME} OK
  $RENAME TEMP AS {FILENAME} OK
  IF CS_CODE¬=0
    WRITE "Renaming of {FILENAME} unsuccessful"
    EXIT
  ENDIF
  READ FILENAME EOF->ENDF
ENDLOOP
>ENDMACRO
```

In this case, the loop will continue to execute until the Boolean variable ENDF becomes TRUE as a result of an end-of-file on input. Also, the clause WHILE NOT ENDF could have been used instead of UNTIL ENDF in the above example.

TESTING AND DEBUGGING MACROS

The current definition of a particular macro can be displayed by issuing the command

>DISPLAY MACRO=macro-name

The display will list each of the statements of the definition along with its line number. The following example illustrates the output from a macro display:

```
#>DISPLAY MACRO=COMPACT
#MACRO COMPACT
#   1: DEFINE FILENAME
#   2: DEFINE ENDF=FALSE
#   3: IF FILE("TEMP") EXISTS
#   4:   WRITE "File TEMP already exists"
#   5:   EXIT
#   6: ENDIF
#   7: READ FILENAME EOF->ENDF
#   8: IF ENDF, EXITLOOP
#   9: LOOP UNTIL ENDF
#  10:   $DUPLICATE {FILENAME} AS TEMP
#  11:   IF CS_CODE¬=0
#  12:     WRITE "Duplication of {FILENAME} unsuccessful"
#  13:     EXIT
#  14:   ENDIF
#  15:   $DESTROY {FILENAME} OK
#  16:   $RENAME TEMP AS {FILENAME} OK
#  17:   IF CS_CODE¬=0
#  18:     WRITE "Duplication of {FILENAME} unsuccessful"
#  19:     EXIT
#  20:   ENDIF
#  21:   READ FILENAME EOF->ENDF
#  22: ENDLOOP
```

A macro can be checked out on a “dry-run” basis by appending the @CHECK modifier to the modifier name when it is invoked:

macro-name@CHECK par1 par2 ...

When the macro is executed, all lines that would normally be emitted for processing by the macro processor or the system are instead printed on *SINK*, prefixed with the character string “*C_”. For example, the COMPACT macro defined above could be checked out by issuing the following command:

```
COMPACT@CHECK
```

The output from the check-run will appear as follows:

```
#COMPACT@CHECK
?DATA1
#*C_  $DUPLICATE DATA1 AS TEMP
#*C_  $DESTROY DATA1 OK
#*C_  $RENAME TEMP AS DATA1 OK
?DATA2
#*C_  $DUPLICATE DATA2 AS TEMP
#*C_  $DESTROY DATA2 OK
#*C_  $RENAME TEMP AS DATA2 OK
```

The execution of a macro can be traced by issuing the macro command

>SET MACROTRACE=ON

When tracing is in effect, all lines generated by the macro invocation and all lines that have parameter

substitution in them are printed on *SINK*, prefixed by a string of the form

*<macro-name>(macrolinenbr) <flag>

where “macro-name” is the macro name, “macrolinenbr” is the line number of statement in the definition that was executed, and “flag” is “g” for macro-generated lines, “s” for substitution lines, or “xxx” for skippped lines. The following example illustrates the output for a trace of the COMPACT macro:

```
#>set macrotrace=on
#compact
#*COMPACT(1)g DEFINE FILENAME
#*COMPACT(2)g DEFINE ENDF=FALSE
#*COMPACT(3)g IF FILE("TEMP") EXISTS
#*COMPACT(4)g XXX   WRITE "File TEMP already exists"
#*COMPACT(5)g XXX   EXIT
#*COMPACT(6)g XXX ENDIF
#*COMPACT(7)g READ FILENAME EOF->ENDF
?data1
#*COMPACT(8)g IF ENDF, EXITLOOP
#*COMPACT(9)g LOOP UNTIL ENDF
#*COMPACT(10)g   $DUPLICATE {FILENAME} AS TEMP
#*COMPACT(10)s   $DUPLICATE data1 AS TEMP
#  $DUPLICATE data1 AS TEMP
 File "DATA1" has been duplicated as "TEMP"; ...
#*COMPACT(11)g   IF CS_CODE¬=0
#*COMPACT(12)g XXX      WRITE "Duplication of {FILENAME} ...
#*COMPACT(13)g XXX      EXIT
#*COMPACT(14)g XXX   ENDIF
#*COMPACT(15)g   $DESTROY {FILENAME} OK
#*COMPACT(15)s   $DESTROY data1 OK
#  $DESTROY data1 OK
 File "DATA1" has been destroyed.
#*COMPACT(16)g   $RENAME TEMP AS {FILENAME} OK
#*COMPACT(16)s   $RENAME TEMP AS data1 OK
#  $RENAME TEMP AS data1 OK
 File "TEMP" has been renamed as "DATA1".
#*COMPACT(17)g   IF CS_CODE¬=0
#*COMPACT(18)g XXX      WRITE "Renaming of {FILENAME} ...
#*COMPACT(19)g XXX      EXIT
#*COMPACT(20)g XXX   ENDIF
#*COMPACT(21)g   READ FILENAME EOF->ENDF
?{end-of-file}
#*COMPACT(22)g ENDLOOP
```

When macro tracing is in effect, all macro invocations are traced. Macro tracing can be combined with the the @CHECK modifier to produce a trace of a check-run. Macro tracing may be disabled by issuing the command

>SET MACROTRACE=OFF

A macro that has been defined can be edited (redefined) by the MTS file editor:

$EDIT
EDIT MACRO macro-name
...
...

STOP

Once the file editor has been invoked for the macro, all of the usual edit commands (including the visual mode) can be used to alter the definition of the macro (only the statements can be edited, not the prototype line). The changes take effect immediately, that is, the macro is redefined as it is edited.

Note that if a defined macro is edited by the above sequence, any copies of the macro that are contained in a macro library are not changed.

MACRO LIBRARIES

Macro definitions may be saved in a macro library which is a file containing several macro definitions and a directory in a special format.

Once a macro library file is attached (established), the macro processor will define the macros contained in the file as they are needed, that is, the definition of a macro will be retrieved from the file when the user invokes it for the first time. Until the macro is invoked, it will remain undefined. This is particularily efficient when the user has a file containing a large number of macros, but only needs a few at a time.

A macro library file is attached by issuing the command

>SET VAR MACLIB(n)="filename"

where the subscript “n” is an index specifying the ordinal number of macro library being attached and “filename” is the file containing the library. The subscript allows more than one macro library to be attached at the same time. Normally, “n” will be 1 if only one macro library is being used.

The format of a macro library file is as follows:

The directory starts at line 1. Each line of the directory has the name of a macro and the line number of the file at which its definition starts. The line number is separated from the macro name by one or more blanks and must be an integer. The directory is terminated by a /END line.

The remainder of the file contains the macro definitions. Each definition begins with the MACRO prototype line and ends with an ENDMACRO line.

The following example illustrates the composition of a macro library file named MACDEFS, which contains definitions for the PAGEPR, PASCAL, and COMPACT macros.

```
   1     PAGEPR 1000
   2     PASCAL 1100
   3     COMPACT 1200
   4     /END
1000     MACRO PAGEPR file COPIES=1
1001     LOOP FROM 1 TO COPIES
1002       $RUN *PAGEPR SCARDS={file} PAR=PORTRAIT
1003     ENDLOOP
1004     $RELEASE *PRINT*
1005     ENDMACRO
1006
1100     MACRO PASCAL source object
```

```
1101     $RUN *PASCALVS SCARDS={source} SPUNCH={object}
1102     IF RUNRC > 0
1103       DEFINE ANSWER
1104       WRITE "Compilation error"
1105       WRITE "Enter OK to continue"
1106       READ ANSWER
1107       IF UPPERCASE(ANSWER)¬="OK", EXIT
1108     ENDIF
1109     $RUN {object}
1110     ENDMACRO
1111
1200     MACRO COMPACT
1201     DEFINE FILENAME
1202     DEFINE ENDF
1203     IF FILE("TEMP") EXISTS
1204       WRITE "File TEMP already exists"
1205       EXIT
1206     ENDIF
1207     LOOP
1208       READ FILENAME EOF->ENDF
1209       IF ENDF, EXITLOOP
1210       $DUPLICATE {FILENAME} AS TEMP
1211         IF CS_CODE¬=0
1212         WRITE "Duplication of {FILENAME} unsuccessful"
1213         EXIT
1214       ENDIF
1215       $DESTROY {FILENAME} OK
1216       $RENAME TEMP AS {FILENAME} OK
1217         IF CS_CODE¬=0
1218         WRITE "Renaming of {FILENAME} unsuccessful"
1219         EXIT
1220       ENDIF
1221     ENDLOOP
1222     ENDMACRO
```

Small macro libraries can be constructed by hand using the file editor. For larger macro libraries, the user can run the program MAC:LIBGEN, which is described in the reference documentation. By convention, the macro definitions start on line 1000, but this is not a strict requirement. The only requirement is that the first definition start after the end of the directory. Most macro generation programs including MAC:LIBGEN start the directory at line 1000 so that there is sufficient room to expand the directory without reformatting the file.

The above macro library may be attached by issuing the command

```
>SET VAR MACLIB(1)="MACDEFS"
```

The above macro library may be revoked by issuing the command

```
>SET VAR MACLIB(1)=""
```

When a macro library is revoked, all of the macros that were defined from the library become undefined.

The MACLIB subscript may be used to attach several macro libraries at the same time. For example, the commands

```
>SET VAR MACLIB(1)="MACDEFS1"
>SET VAR MACLIB(2)="MACDEFS2"
```

```
>SET VAR MACLIB(3)="MACDEFS3"
```

will attach the macro libraries in the files MACDEFS1, MACDEFS2, and MACDEFS3. If a macro of the same name is present in more than one macro library, the definition is taken from the macro library attached with the subscript.

A macro library file that has been attached by the "SET VAR MACLIB" command may be edited by the file editor. However, for any of the macros in the file that are currently defined, their current in-line definitions remain unchanged regardless of any changes made to the library file. The revised definitions in the file may be made the current definitions by reissuing the SET command.

SYSTEM MACRO LIBRARY

One macro library is always supplied by default, and is attached to MACLIB at subscript 0 so it will be searched last. This macro library is in the file *CMDMACLIB. It currently contains only one macro, MACROLIB, which is used to attach other macro libraries. This macro is more convenient than the notation is given earlier because it keeps track of the indices and requires less typing; it offers no more power. Thus, instead of typing in

```
SET VAR MACLIB(1)="MACDEFS"
```

one can just say

```
MACROLIB MACDEFS
```

In addition,

```
MACROLIB LIST
```

will produce a list of the libraries currently attached.

REFERENCE SECTION

GENERAL COMMENTS

The items mentioned herein are available only if the MTS SET command option MACRO has been set ON:

```
$SET MACRO=ON
```

Note that all other SET commands in this document are macro SET commands, not MTS SET commands.

I/O STREAM MACRO PROCESSOR OVERVIEW

One approach used in describing the macro processor is to present it as an automatic $SOURCE-file generator. For many standard operations $SOURCE-files are good enough in MTS. However $SOURCE-files are not very flexible. Say, for example, you have a repetitive process that you have to go through for a number of different files: Update file A, producing file B, compile it into file C, then linkedit into file D. You cannot write a "generic" $SOURCE-file that will work for any set of files (A,B,C,D). Or say you would like to run program X, then depending on the outcome run either program Y or Z. None of these things work well with $SOURCE-files, and that is the problem the Macro Processor is meant to solve. It creates the "$SOURCE-file" line-by-line, on the fly, as successive lines are needed from *SOURCE*, using the instructions you give it in the form of a macro definition.

Without diving into the internal details of MTS too much, another useful way to view the macro processor is as a black box sitting just behind *SOURCE*. In the normal course of events, data is read from *SOURCE* in many ways: MTS reads commands from *SOURCE*; CLSs read commands from *SOURCE*; user programs also get data from *SOURCE* either explicitly (3=*SOURCE*) or implicitly (SCARDS defaults to *SOURCE*). But if the macro processor is active it takes a look at everything coming in from *SOURCE*. Depending on the situation it may transform lines, completely swallow lines, or produce new lines.

If the macro processor is activated, it gets first crack whenever anybody requests data from *SOURCE*. If it is in the middle of a macro expansion it just conjures up a line of data and gives it to MTS which returns it to whoever requested the data in the first place (i.e., READ, SCARDS, etc.). If the macro processor did not have anything ready, MTS goes ahead and gets a data line from whatever *SOURCE* is attached to. Usually this is the user's terminal, but it might be a file, tape, etc. Then the macro processor gets another crack. It takes a look at the line the user just read in and does a number of things. If it is the beginning of a macro definition (e.g., >MACRO QQSV) the macro processor swallows the line and reads in the rest of the macro definition, then tells MTS to get another line of data before returning. If it recognizes the line as a macro invocation (e.g., QQSV) it replaces the data with the first expansion line and returns that to the user. The remaining lines are then returned in subsequent *SOURCE* calls.

If none of the above made sense, maybe an example will help. Here is an example of the definition and use of a simple macro. Appendix F of this document describes in nitty-gritty detail what MTS goes through to do this. It will be discussed at a higher level here:

```
# set macros=on
# >macro hasplog
? $run unsp:hasplog
? l u w164
```

```
? stop
? >endmacro
#
# hasplog
# Execution begins
:  JOB: 674553  (4917317)  USERID:W164 S 07-08-82
:  JOB: 674559  (4917323)  USERID:W164 S 07-09-82
# Execution terminated
```

Here macro HASPLOG is defined to run program UNSP:HASPLOG. This example is so trivial that it could just as well be done with a simple $SOURCE-file, but hopefully you will get the idea.

In the second line, MTS was prompting for an MTS command with a "#"—that is, it called SCARDS (more or less) to get a new command line. But the macro processor recognized the ">macro hasplog" line (see the section, "Macro Flag Character", for an explanation of the ">") as something IT needed, so that line never made it back to MTS. (The process is similar to what happens if you stick a "%" device command on the front of a line.) The macro processor then proceeded to swallow up everything through the ">endmacro" line. The line finally given back to MTS was the null line that was typed immediately after the ">endmacro" line.

The "hasplog" line illustrates the reverse, macro expansion, process. MTS called "SCARDS", and "hasplog" was typed—but that never made it back to MTS. The macro processor recognized "hasplog" as something IT wanted, swallowed it, and gave back in its place the "$run unsp:hasplog" line of the HASPLOG macro. On subsequent calls to "SCARDS", the macro jumped in and supplied "l u w164" and "stop" without the user ever being asked to type in anything on the terminal.

THE EXPLAIN FACILITY

There is an EXPLAIN command which will provide information about the command extensions.

Syntax

```
EXPLAIN [phrases]
```

When you enter

>EXPLAIN

you begin talking to an explain program. This is indicated by the input prefix, which becomes "Explain?". If your initial line had no parameters on the command, you get a general explanation of the explain facility. If you gave one or more phrases, it looks them over to see if it can recognize them as something it has an explanation for, and then starts with that explanation. If you enter a null line at any time, it continues with the next paragraph of explanation of whatever it was presenting, or if it had finished that, it goes on to the "next" thing. Entering STOP or an end-of-file will get you out of the explain processor.

MACRO FLAG CHARACTER

In most cases, most lines that are to be inspected by the macro processor for macro invocations or macro commands must be prefixed with a flag character, which is ">". This includes substitution: only lines that are macro commands or macro invocations are inspected for substitution (as signalled by {}).

The rules are currently as follows:

- If a line has a > in column 1 and not in column 2, the line is destined for the macro processor and must contain a valid macro invocation or macro command. (Note here et seq. that if the line is being read in MTS command mode, edit command mode, or similar, then any optional leading $ is removed first, and then the resultant column 1 ... is inspected.)

- If a line has > in both columns 1 and 2, then the leading > is removed and the remainder of the line is transmitted unchanged.

- Lines coming from a macro expansion are normally inspected for substitution, macro invocation, and macro commands. Such line do not require a flag character but may have one.

- Lines not coming from an expansion (as for example, lines typed in at the terminal) that are to be interpreted as macro commands must always have a flag character.

- Lines not coming from an expansion, but being read by a command processor (MTS or EDIT, etc.) that are to be interpreted as a macro invocation may have a flag character, but do not have to.

- Lines not coming from an expansion and not being read by a command processor (for example, lines read by a program that is getting input from the source stream) that are to be interpreted as a macro invocation must normally have a flag character.

See Appendix A for a description of the @MACRO and @MFR modifiers that control this.

See Appendix B for an example of interaction between MTS command flag characters "$", macro command flag characters ">", and various CLSs (MTS and EDIT).

ERROR MESSAGES

All messages produced by the "macro facility package" start with the three characters "*>*" in order to clearly identify who is issuing the message.

VARIABLES AND EXPRESSIONS

These are items for the command language which can occur outside macros and which should appear to the user as independent of macros.

All of the commands described in this document are macro commands, not MTS commands, and hence may need to be prefixed with a macro flag character as described in the previous section.

The items described in this chapter consist of commands and expressions that may appear as parameters to those commands. Expressions may also be used to generate parts of emitted MTS commands. Since the most common form of an expression is a simple variable name, variables are discussed first. Then general expressions are covered, and the finally commands that use variables and expressions.

VARIABLES

There are two types of variables, *user* variables and *system* variables.

User and system variable names are comprised of 1 to 255 alphanumeric characters with the first character being alphabetic. Here "alphabetic" means only the uppercase alphabet, to which is added the underscore character "_". Lowercase letters are automatically converted to uppercase.

Defining a Variable

All user variables must be defined before they are used. Normally, this is done explicitly although some some user variables are automatically defined when a macro is invoked (see the section, "Macros"). System variables always exist and do not need to be defiined.

The command for defining a simple user variable is:

Syntax

```
DEFINE var[=initvalue] [attributes]
```

It is illegal to define a variable that already exists. The optional "attributes" may include:

GLOBAL
CONSTANT

Without any attributes, the DEFINE command defines a local variable whose scope is at the current level. That is, if given outside of a macro expansion, it is a global variable; if given within a macro, it is a variable local to that macro.

Types

Variables are automatically typed (as, for example, is done within the SNOBOL programming language). This means that as the type of the variable changes (e.g, from string to integer) the system keeps track of the change. What happens when an operator is applied to data items depends on the type of the values involved, and not on the type associated with a variable name as in typed programming languages. Internal types of the variables include character string, integer, Boolean, and line number. All command-language communication is done in character strings.

Aggregates: Arrays and Structures

An aggregate is a collection of multiple data items associated with one name. There are two basic types of aggregates available. One is the array, either indexed by integers in the traditional manner, or used as an associative array, where the selecting value is arbitrary instead of being restricted to integers. (This latter form is called a "table" in SNOBOL.) The other type of aggregate is a one-level structure, which may be used either in a static manner or may be used to create dynamic instantiations of the structure form.

Arrays

Arrays are specified in the traditional manner. The form of the definition is:

Syntax

```
DEFINE array-name(dimension1,dimension2,...)
```

where each of the items in the list separated by commas specifies the form for that dimension of the array. The possible forms for each dimension are

(1) null	Example: DEFINE A()
(2) integer	Example: DEFINE A(10)
(3) integer...integer	Example: DEFINE A(0...10)
(4) string-constant	Example: DEFINE A("NAME")

The first case specifies that that subscript must be numeric but no bounds checking is done. The second case specifies that the subscript must be an integer between 1 as the low value and the number given as the upper value. The third case specifies that the subscript must be an integer between the two values given. The fourth case specifies that the subscript may be anything (the value of the string constant should have some mnemonic significance to whatever use is to be made of that subscript, but does not affect the definition otherwise).

One-dimensional arrays of cases 1, 2, or 3 above, and with lower subscript 1, will be automatically converted to and from strings as required.

Structures

Structures consist of a fixed number of named fields. The form of definition is:

Syntax

```
DEFINE structure-name(field1,field2,...)
```

There must be at least one field name. Each of the field names must be different within one structure definition, but a given structure definition may have a field with the same name as another structure definition. But while a field may have the same name as a field of a different structure, it may not have the same name as any other object.

Static Use

Each DEFINE statement creates a structure of that name. The fields of this structure are referenced in the form "structure-name:field-name". For example,

```
DEFINE NAME(FIRST,LAST)
```

defines a structure NAME with two fields, FIRST and LAST. Values may be set and extracted, for example,

```
SET VAR NAME:FIRST="Hortense"
SET VAR NAMELENGTH = SIZE(NAME:FIRST)
```

Dynamic Use

Additional copies of the structure may be obtained by using the structure name as a function with arguments being initial values for the fields. The value of this function is a copy of that structure with fields as specified, and this copy may be accessed using the ":" operator. For example,

```
DEFINE NAME2=NAME("John","Psmith")
```

will make a NAME and it can be used

```
DEFINE INVERTED = NAME2:LAST || ", " || NAME2:FIRST
```

Named Constants

Named constants can be defined by specifying CONSTANT as an attribute on the DEFINE command. In this case, an initial value is required. This is the same as an initialized variable except that its value can not be changed. The variable can, however, be undefined with the FORGET command. For example,

```
DEFINE ON=1 CONSTANT
DEFINE OFF=0 CONSTANT
```

Scope

The "scope" of variable specifies "under what cases is which definition of a variable used." In other words, in how many different cases can the same variable be defined. There are three categories, user global, user local, and system, in which a variable may appear. User local means it was defined inside a macro definition. A given variable name may have several local definitions due to its use at several levels of macro call, but only the definition in the currently active macro level, if any, is accessible.

Order of Resolution

The following precedence of variable scope occurs, with top to bottom being first to last:

(1) local user
(2) global user
(3) system

Overriding the Default

If there is a local version of a variable, the global version is hidden until the local one is eliminated. However, the system definition may be specified instead of the user definition by attaching the modifier @SYSTEM to the variable reference.

Disposing of Variables

The definition of a variable may be removed with the command

Syntax

```
FORGET variable
```

Local variables are automatically undefined as the macro level pops at the termination of a macro's expansion. Changes of mode from one CLS to another do not affect any variables.

Using Variables

Both system and user variables may be used at any point in a macro command, as well as at any point in an MTS command emitted by a macro. In emitted MTS commands, a variable reference must be indicated by surrounding it with the substitutable-quantity-delimiters (SQD) which are preferentially the braces "{}", for example

```
$RUN PROG SCARDS={INPUT} SPRINT=*PRINT*
```

These delimiters may be used as well in macro commands where symbolic substitution is wanted. However, normally in macro commands the parameters are treated as expressions, so variables may just be used as normally in expressions.

Alternatively, the form "&(var&)" may be used in place of "{var}".

```
$RUN PROG SCARDS=&(INPUT&) SPRINT=*PRINT*
```

The construct "&(" is treated as syntactically equivalent to "{", and "&)" is equivalent to "}", so the left and right delimiters can be of different notations.

The substitution is a string substitution, that is, it acts as if the characters from the left delimiter through the right delimiter were physically replaced by the contents of the variable (converted to characters if necessary).

The scan for substitution goes left to right, and after each item is substituted, continues with the first character after the last one substituted (i.e., the default is no rescan).

The substitution can be nested, and the evaluation is done from inside to outside.

Substitution is done before the line is ever submitted to whatever is inspecting its content, and so, in particular, before string delimiters are recognized. Thus, putting quotes around a brace does not protect it from being treated as a substitution delimiter. To get a literal brace in a line, it must be doubled.

Setting Values

Command variables can be assigned values via the command language by using the SET command:

```
Syntax──────────────────────────────────────────────────────────────────────────┐
│                                                                               │
│  SET VAR var=expr                                                             │
│                                                                               │
└───────────────────────────────────────────────────────────────────────────────┘
```

Note that because a name rather than a value is expected to the left of the "=" in a substitution, SQDs are not placed on the variable reference. Also, because this is a case where an expression is expected, then as is shown in the next section, the SQDs on any variables in the expression may be omitted. Thus, it is possible to write

```
SET VAR I=I+1
```

as well as

```
SET VAR I={I}+1
```

Displaying Values

The values of variables and expressions may be displayed by using the DISPLAY command:

```
Syntax──────────────────────────────────────────────────────────────────────────┐
│                                                                               │
│  DISPLAY ARRAY(var)  or  DISPLAY ARRAY=var                                    │
│  DISPLAY FUNCTIONS                                                            │
│  DISPLAY MACLIB(index)                                                        │
│  DISPLAY MACLIB("name")                                                       │
│  DISPLAY MACRO(var)  or  DISPLAY MACRO=var                                    │
│  DISPLAY MACROS                                                               │
│  DISPLAY STRUCTURE(var)  or  DISPLAY STRUCTURE=var                            │
│  DISPLAY SYMBOLS                                                              │
│  DISPLAY SYSTEM_VARIABLES                                                     │
│  DISPLAY USER_VARIABLES                                                       │
│  DISPLAY VALUE(expr)  or  DISPLAY VALUE=expr                                  │
│  DISPLAY VARIABLES                                                            │
│                                                                               │
└───────────────────────────────────────────────────────────────────────────────┘
```

If "var" is the name of an array, its prototype and all non-null elements may be displayed with

```
DISPLAY ARRAY(var)
```

To display the names of user-defined functions that are defined in-line (i.e., not from macro libraries),

```
DISPLAY FUNCTIONS
```

A list of the macro names in a macro library may be displayed either by using its MACLIB index or its name, that is,

```
DISPLAY MACLIB(index)
```

or

```
DISPLAY MACLIB("name")
```

To find out what macro libraries are active, display the contents of the array MACLIB,

```
DISPLAY ARRAY(MACLIB)
```

To display the names of macros defined in-line (i.e., not from macro libraries),

```
DISPLAY MACROS
```

If "var" is the name of a macro, its definition may be displayed with

```
DISPLAY MACRO(var)
```

If "var" is the name of a structure or structure-instance, its prototype and the values of its fields may be displayed with

```
DISPLAY STRUCTURE(var)
```

To display the names of system-defined symbols,

```
DISPLAY SYSTEM_VARIABLES
```

To display the names of user-defined symbols (this includes macros),

```
DISPLAY USER_VARIABLES
```

To display both user-defined and system symbols,

```
DISPLAY VARIABLES
```

If "var" is the name of a variable, its value may be displayed with

```
DISPLAY VALUE(var)
```

In addition, all of the above plus the meta-command names and the operators (usually a long list) may be displayed with

DISPLAY SYMBOLS

EXPRESSIONS

An expression is a formula indicating how a value is to be computed. There are three kinds of expressions. The simplest object that is an expression is a constant, since the object is just the value written out. The next simplest is a variable, and the value is its contents. The last kind of expression is one that produces a value from other (simpler) expressions by combining them. These combinations are either composed of an operator and one or two operands, or else a function-name and zero or more operands, depending on the function-name involved. In this latter case, the operator is really that of function-call.

An expression, as defined in this section, may occur in macro commands (such as IF) described later in this volume, and inside the braces in a {...} construct.

Unlike the MTS command language, in an expression blanks may be used between items to improve readability, since other syntactic constructs are used to parse things. However, blanks may usually be left out if wished.

Operators

The operators are largely borrowed from "standard" programming languages. They may be grouped according to the types of objects they operate on.

(1) Numeric Operators:

These take integers or line numbers, or character strings whose contents express an integer or line number.

Addition: +
Subtraction: – [binary]
Multiplication: *
Division: /
Negation: – [unary]

The operands for addition, subtraction, multiplication, and division must be integers, line numbes, or strings whose contents can be converted to integers or line numbers. The result for addition, subtraction, and multiplication is an integer if both operands were integers; otherwise, it is a line number. The result for division is always a line number (thus, 3/2 is 1.5).

The unary "–" produces the negation (the same value but opposite sign) of its operand.

(2) String Operators:

Concatenation: | |
Substring: string(beg**...**[end]) or string(beg | len)

Both operands of the concatenation operator must be strings or capable of being converted to strings. The result is a string whose contents are the value of the first

operand followed by the value of the second operand.

The substring operation is a specialized form of subscription. There are two ways of specifying:

string(beg**...**end)

where "beg" and "end" are the beginning and ending subscripts for the substring desired, assuming the first character of "string" has subscript 1, and

string(beg**|**len)

where "beg" is again the beginning subscript, and "len" is the length of the substring desired. If the rest of the string starting at "beg" is wanted, the "end" may be omitted from the first form:

string(beg**...**)

The beginning and ending subscripts must be integers.

(3) Comparatives:

Less-than: <
Equal-to: = or IS
Greater-than: >
Less-than-or-equal-to: <=
Not-equal-to: ¬= or ISNT
Greater-than-or-equal-to: >=
[The "Fortran" form of writing the comparatives (**.**LT**.** et al.) is also available.]

If both operands are numeric, a numeric comparison is done. If one of the operands is numeric and the other operand is a string which can be successfully converted to a number, it will be so converted and the comparison will proceed numerically. If both operands are strings and both can be converted to numbers, both will be converted and the comparison will proceed numerically. Otherwise, a string comparison will be done. The result of these operators is always Boolean (true or false).

The forms IS and ISNT are provided for readability, so one may write "IF ZNORF IS 4". The IS and ISNT forms require one or more blanks both before and after the operator name; if this is not desired or not allowable, the dotted forms **.**IS**.** and **.**ISNT**.** may be used.

For additional ease in writing "readable expressions" in conjunction with the FILE and VARIABLE built-in functions, the following are provided:

DOES is same as IS,
DOESNT is the same as ISNT,
The EXIST operand is the same as NONNULL, and
The EXISTS "operator" is the same is as IS NONNULL.

NONNULL is a special version of the system variable NULL which causes the operator it is associated with to be reversed. Thus, IS NONNULL becomes ISNT NULL. All of which leads to "... EXISTS" means "... ISNT NULL" and "... DOESNT EXIST" means

"... IS NULL". See the description of the built-in functions FILE and VARIABLE for examples of use.

(4) Logical Connectives:

AND
OR
NOT

The AND operator performs the logical "and" of its two operands. Both operands must be Boolean (i.e., true or false), and the result is Boolean. The result is true if and only if both operands are true.

The OR operator does the logical "or" of its two operands. Both operands must be Boolean (i.e., true or false), and the result is Boolean. The result is true if and only if one or both operands are true.

The NOT operator does the logical "not" of its operand. The operand must be Boolean (i.e., true or false), and the result is Boolean. The result is true if and only if the operand is false.

Whenever an AND or OR operator is evaluated, the left operand is evaluated first. If this determines the value of the operation (since if the left operand of an AND is false the value will be false, or if the left operand of an OR is true the value will be true), then the right operand is not evaluated. This is usually referred to as "McCarthy evaluation" or "McCarthy AND and OR". This allows writing such expressions as

```
IF VARIABLE("A") EXISTS AND A=42, ...
```

whereas otherwise one would have to write

```
IF VARIABLE("A") EXISTS
   IF A=42, ...
ENDIF
```

(5) Substitution:

=
+=
−=
||=

Each of these operators takes a left-hand side which is a name and a right-hand side which is an expression.

The "=" (substitution) operator stores the value of the expression in the item the name specifies.

The "+=" (plus-and-becomes) operator is similar to the substitution operator except that the right-hand side is added to the current value of the left-hand side. Thus, a statement of the form

name += expression

is treated as if it was

name = name + (expression)

The "–=" (minus-and-becomes) operator is similar to the substitution operator except that the right-hand side is subtracted from the current value of the left-hand side. Thus, a statement of the form

name –= expression

is treated as if it was

name = name – (expression)

The "||=" (append) operator is similar to the substitution operator except that the right-hand side is concatenated onto the end of the current value of the left-hand side. Thus, a statement of the form

name ||= expression

is treated as if it was

name = name || (expression)

In substitution cases (i.e., SET VAR v=e), only the leftmost "=" at zero-level nesting depth (with respect to parentheses) is treated as substitution; all others at zero level are treated as comparatives. Thus,

```
SET VAR A=B=C
```

sets variable A to *true*, if B=C, and *false* otherwise.

(6) Function Call and Array Subscription:

The function call operation and the array subscription operation are both expressed in the form of "variable followed by parenthesized list". If an element of an array is itself an array, the elements of the second array may be accessed via another parenthesized list of subscripts following the first, as for example A(5)(3) (which note is not the same as A(5,3)).

A function call is signalled by a left parenthesis that is immediately preceded by a variable name where the value of that variable is a built-in function or a user-defined function. The parameters to this function call form a list of expressions, separated by commas, between the left parenthesis and its matching right parenthesis. If there are no parameters, the parentheses are still needed. The result of this operation is the value returned by the function.

Subscription is signalled by a left parenthesis that is immediately preceded by a variable name where the value of that variable is an array or something that can be converted to an array. The subscripts to this array reference form a list of expressions, separated by commas, between the left parenthesis and its matching right parenthesis. There must

be at least one subscript given. The result of this operation is the selected array element. If that array element is itself another array, then another list of subscripts, surrounded by parentheses, may be appended to select the element of that array, and so forth. Finally, a "subscript" which is a range specification may be given to produce a substring.

(7) Field Selection:

:

The colon operator is used to select a field from a structure or a structure-copy. The left operand must have a structure as value and the right operand must be the name of a field of that structure. The result is the selected field of that structure.

Variables

A command variable, when used "normally" as a variable in an *expression* may be given without the substitutable quantity delimiters (SQDs), although the delimiters may be used. If, however, the value of a variable is to be used to construct some or all of some item that is to be subsequently evaluated in the expression, then the variable must be delimited with the "{}" (or equivalent).

In other words, in an expression there are two stages of processing. First, all substitutable quantities are substituted in as a string substitution. Then what is left is evaluated as an expression.

Note that substitution processing of a line is done *before* it is ever inspected for content, and thus any variables whose values are to be substituted into the line must be defined and have values before the line is processed. On the other hand, expression evaluation occurs (if it occurs) during processing of the line. Thus,

```
LOOP FOR I FROM 1 UNTIL PAR_{I}>3
```

will not do what the writer probably wanted. For this case, the command will have to be broken up into two commands, with

```
LOOP FOR I FROM 1
```

as the first command, and then

```
IF PAR_{I}>3, EXITLOOP
```

as the second command.

In expressions, " and { are approximately opposites. Thus, if ZVR is a variable,

```
WRITE ZVR
```

and

```
WRITE "{ZVR}"
```

usually do the same thing. However, if the contents of ZVR are A"B, then

```
WRITE ZVR
```

will successfully write out A"B, whereas

```
WRITE "{ZVR}"
```

becomes after substitution

```
WRITE "A"B"
```

which is invalid.

Note also that outside of expressions the "{...}" construct is not unexpected, since in that case {...} is needed to force an expression.

Constants

These are constants for the new constructs. However, these are intended to blend in with constants as used in existing MTS commands.

(1) Character Constants

Character constants consist of a string of delimited characters. The delimiter is the double-quote (") or the prime ('). The terminal delimiter of the string must be the same character as the initial delimiter, and if this character is to be represented in the body of the string, it must be doubled.

The delimiters are merely for grouping. They do not prevent substitution as flagged by the "{ }" or "&(" or "&)". To get the character "{" in a string you must double it ("{{"); likewise for "&(" and "&)". (Note: An ampersand followed by a character that is not a parenthesis is just an ordinary character and not a delimiter.)

(2) Integer Constants

Integers, as usual, consist of digits optionally preceded by "+" or "–". The resultant number must be in the range –9223372036854775808 to 9223372036854775807.

(3) Line-Number Constants

Line-number constants follow the usual conventions for numeric line numbers: a number in the range –2147483.648 to 2147483.647 with up to three digits after the decimal point.

(4) Boolean Constants

The Boolean constants are TRUE and FALSE.

(5) Hexadecimal Input

The hexadecimal form of a constant may be entered as X'...' or X"...". It is considered as "typeless" until it is combined with operands of other types (or used in operations that are type-specific), at which point it is treated as the type concerned. Use of hexadecimal input form is not recommended unless you are knowledgeable about the internal form of the various data types.

(6) Array Denotations

One-dimensional arrays with fixed-bounds (or unbounded) and a lower-bound of 1 can be expressed as a string whose contents are a parenthesized list with the elements separated by commas, for example, "(A,B,C,(D,E))". In this example, the fourth element is itself another array.

Built-In Functions

Built-in functions are system-provided entities that take one or more arguments and produce a result. They are invoked by using

function-name(arg1,arg2,...)

in an expression where a constant or variable might be used. "function-name" is the name of the function; "arg,..." are the arguments, separated by commas and the whole argument list enclosed in parentheses. For certain built-in functions, a final "argument" of the form "RC–>variable" may be specified to cause a return code to be stored instead of a error message occurring in the case that an error condition was detected. See the specific function descriptions for details.

The built-in functions provided are:

ABS

Syntax

```
number = ABS(number)
```

ABS returns the absolute value of its argument.

ABSTIME

Syntax

```
absolute time = ABSTIME(integer)
absolute time = ABSTIME(string)
```

For integer arguments, ABSTIME takes the value of its argument as the number of microseconds since 00:00:00 March 1, 1900, local time. If the argument is a string, it is assumed to be the character representation of a time/date. In both cases, the value is returned is of type "absolute time". For example,

```
"2:35 pm Mar 7, 1985"
"NOW"
"12-06-79"
```

See Appendix G for a description of the legal forms of a time/date.

ARRAY

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  array = ARRAY(string)                                                    │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

ARRAY takes a string as argument and returns an array as a result. The string's contents should be the same as what would be between the parentheses in an equivalent DEFINE command. For example, SET VAR X = ARRAY("") will set variable X to an unbounded one-dimensional array.

CAN_CONVERT

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  Boolean = CAN_CONVERT(value,string)                                      │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

CAN_CONVERT returns TRUE if "value" can be converted as requested by "string". Legal values for "string" are:

INTEGER
LINENUMBER
NUMBER
STRING
ARRAY

For example,

```
SET VAR X = CAN_CONVERT(Y,"INTEGER")
```

CONTROL

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  result = CONTROL(string,string)                                          │
│  result = CONTROL(fdubptr,string)                                         │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

CONTROL takes two arguments: if the first argument is a string, it is assumed to be an FDName and a new FDUB is obtained for this call to the CONTROL subroutine and released afterwards. A "fdubptr" is something acquired from the FDUB–>x construct on a READ or WRITE command. The second argument is the string passed to the CONTROL subroutine.

The result is 0 if the CONTROL subroutine gave return code 0. If the subroutine returned a non-zero return code, the DSR return code is returned as value if no message string was available. If both DSR return code and DSR message were returned, the value is the string "(n,message)" which is a list containing the two items. This allows callers to test, for example,

given

```
SET VAR X=CONTROL("*MSOURCE*",CMD)
```

one can test the result

```
IF X(1)=0
```

and then either print the value of X, or test the DSR return code in X(1) and optionally do something with the DSR message in X(2) if it exists (use MAX_SUBSCRIPT).

Note that since macro built-in functions cannot change the value of their parameters, this macro CONTROL cannot be used to get the sense data. For this, the SENSE built-function must be used.

COST

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  integer = COST("CC")                                                     │
│  line-number = COST("$")                                                  │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

COST returns the cost of the current session as obtained from the system COST subroutine. The first form returns the quantity in integer centicents (one-hundredth of one cent); the second form returns it in a “dollars.cents” form, rounded to the nearest cent, as a line-number.

CUINFO

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  old-value = CUINFO(item,new-value[,RC->variable])                        │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

CUINFO takes two arguments. The first argument is either integer-valued (specifying the index) or string-valued (specifying the name). (Note: A numeric string, such as "23", is considered integer-valued in this case.) The system subroutine GUINFO is then called with this argument, and the “old-value” is the value returned by this call. Then the system subroutine CUINFO is called, for the same item, giving it the “new-value”.

The value depends on which item is asked for, and is usually string or integer. Old values are returned and new values are expected in their internally stored form, as specified in the GUINFO description in MTS Volume 3, *System Subroutine Descriptions*. Thus, for example,

```
CUINFO("RCPRINT",4)
```

is correct, but not

```
CUINFO("RCPRINT","NONZERO")
```

If the system subroutines GUINFO or CUINFO detect an error and the optional “RC–>variable”

has been specified, then the return code from the system subroutine is stored in the variable given (which must be a simple variable, not an element of an array or structure). In this case the value returned by CUINFO is unspecified, so the only safe thing to do is to store it in a variable until the return code can be inspected to see if it is safe to use it. If the system subroutines GUINFO or CUINFO detect an error and there is no "RC–>variable" specification, then an error message is printed and macro processing is aborted.

DOUBLE

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = DOUBLE(string1,string2)                                         │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

DOUBLE takes two arguments: first is the string to be worked on, the second is the string of characters to be doubled. The result is a string which is the same as the first argument, except that every character in it which is also in the second argument is doubled.

DUPL

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = DUPL(string,integer)                                            │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

DUPL takes two arguments: first is the string to be duplicated and the second is the integer count of the number of copies wanted. The result is a string containing the number of copies requested of that particular string.

EVAL

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  value = EVAL(string)                                                     │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

EVAL takes a string as argument, evaluates it as an expression, and returns its value. Thus, if we have

```
DEFINE P2=4
DEFINE J=2
```

compare

```
WRITE "P"||J||"+3"
```

which writes

```
P2+3
```

with

```
WRITE EVAL("P"||J||"+3")
```

which writes

```
7
```

FILE

Syntax

```
string = FILE(string)
```

FILE takes a string as argument and returns the null string if there is no file of that name or returns the file name otherwise. For example,

```
IF FILE("ZMWQ") EXISTS, ...
```

(Note that currently "doesn't exist" means either "doesn't exist" or "you have no access")

FILE_INFO

Syntax

```
value = FILE_INFO(string1,string2[,RC->variable])
```

FILE_INFO takes a string specifying a simple file name as the first argument and a string specifying an item of information from the list below as the second argument, and returns the information requested. Information is obtained by calling the system subroutines GFINFO or CHKFILE.

The items available and the type of information returned are:

OWNER	string	
VOLUME	string	
USECOUNT	integer	
TYPE_NUMBER	integer	value returned by GFINFO
TYPE	string	LINE, SEQ, SEQWL, or ?
LOC_NUMBER	integer	value returned by GFINFO
LOC	string	2311, 2314, CELL, 3330, 3350, 3370, 3280, 3340, 3375, 3380, or ????
NOSAVE	Boolean	
PRIV	Boolean	
PKEY	string	
EMPTY	Boolean	
BKWDOK	Boolean	
SIZE	integer	
TRUNCSIZE	integer	

MINSIZE	integer	
MAXSIZE	integer	
FIRST	line-number	
LAST	line-number	
MAXLEN	integer	
LINES	integer	
HOLES	integer	
USED	integer	
AVAIL	integer	
MAXHOLE	integer	
EXPFAC	integer	
ACCESS	integer	value 0 to 63, bits of access byte
READ_ACCESS	Boolean	
WRITE_ACCESS	Boolean	
READ_WRITE_ACCESS	Boolean	
DESTROY_ACCESS	Boolean	
PERMIT_ACCESS	Boolean	
UNLIM_ACCESS	Boolean	
CREATE	absolute time	
LASTCAT	absolute time	
LASTDATA	absolute time	
LASTREF	absolute time	

The access items are the values returned by CHKFILE for the current signon ID, current project, current pkey combination. For additional information on the other items, see the description of items of the same name for the FILESTATUS command in MTS Volume 1, *The Michigan Terminal System*.

FILES

Syntax

```
string = FILES(string)
```

FILES takes a string which is a "FILESTATUS pattern" as argument and returns a list (i.e., array) of the names that match. For example,

```
LOOP FOR X OVER FILES("?.S")
```

(FILE and FILES are actually two names for the same built-in function, but it seemed easier to present the two aspects separately.)

FIND_CHAR

Syntax

```
integer = FIND_CHAR(string1,string2)
```

FIND_CHAR takes two strings as arguments. It searches the first string for the first character which is also in the second string and returns the integer index at which it was found. If not found, a value of zero is returned.

FIND_STRING

Syntax

```
integer = FIND_STRING(string1,string2)
```

FIND_STRING takes two strings as arguments. It searches the first string for the first occurrence of the second string and returns the integer index at which it was found. If not found, a value of zero is returned.

GUINFO

Syntax

```
value = GUINFO(argument[,RC->variable])
```

GUINFO takes a single argument which is either integer-valued (specifying the index) or string-valued (specifying the name). (Note: A numeric string, such as "23", is considered integer-valued in this case.) The system subroutine GUINFO is then called with this argument. The value depends on which item is asked for, and is usually string or integer. Values are returned in their internally stored form, as specified in the GUINFO description in MTS Volume 3, *System Subroutine Descriptions*. Thus (assuming the default setting), GUINFO("RCPRINT") will return the integer 4, not “NONZERO”, for example.

If the system subroutine GUINFO detects an error and the optional “RC–>variable” has been specified, then the return code from the GUINFO subroutine is stored in the variable given (which must be a simple variable, not an element of an array or structure). In this case the value returned by the function is unspecified, so the only safe thing to do is to store it in a variable until the return code can be inspected to see if it is safe to use it. If the system subroutine GUINFO detects an error and there is no “RC–>variable” specification, then an error message is printed and macro processing is aborted.

IGNORE_CHAR

Syntax

```
integer = IGNORE_CHAR(string1,string2)
```

IGNORE_CHAR takes two strings as arguments. It searches the first string for the first character which is not in the second string and returns the integer index at which it was found. If all characters in the first string are also in the second, a value of zero is returned.

INTEGER_PART

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  integer = INTEGER_PART(number)                                           │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

INTEGER_PART returns the integer part of the argument supplied. If the argument is integer to start with, it is returned as value; if the argument is a line number, the integer part of the line number is returned.

LOCK

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  Boolean = LOCK(string,how)                                               │
│  Boolean = LOCK(fdubptr,how)                                              │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

LOCK takes two arguments, where the first argument is the same as for CONTROL. The value of the second parameter should be a string with one of the values: “READ”,“MOD” or “MODIFY”, “DES” or “DESTROY”, or “OFF”. The value of the function is TRUE if the locking was done as requested. If the locking cannot be done now (LOCK subroutine produced RC=20) the value is FALSE. If the LOCK subroutine detected any error conditions (file does not exist, ...) an error message is printed and macro processing is aborted.

If you want to intercept all return codes from the LOCK subroutine, then a “final parameter” of the form RC–>x must be used. In this case, if the value of the LOCK built-in function is FALSE, then the value of “x” will be the return code from the LOCK subroutine. For example,

```
SET VAR Z=LOCK("BARPHEE","MOD",RC->W)
```

LOWERCASE

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = LOWERCASE(string)                                               │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

LOWERCASE takes a string as an argument and returns that string with all the uppercase letters converted to lowercase.

LPAD

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = LPAD(string,integer[,padstring])                                │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

LPAD requires two arguments: the first is the string to be padded and the second is the integer number of columns wanted for the final result string. An optional third argument is the string to be used for padding. If not given, the pad string defaults to a single blank. If the string is shorter than the number of columns wanted, it is padded on the left with the appropriate characters; otherwise the result string is the same as the argument string.

LTRIM

```
Syntax
  string = LTRIM(string)
```

LTRIM returns as a value the argument string, trimmed of all leading blanks.

MAX

```
Syntax
  number = MAX(number1,number2)
```

MAX returns as a value the largest of its two arguments.

MAX_SUBSCRIPT

```
Syntax
  integer = MAX_SUBSCRIPT(array)
```

MAX_SUBSCRIPT takes a fixed-bounds array as argument and returns the maximum value for the first subscript as integer value. If given an unbounded array, the value returned is the largest subscript set so far.

MICROSECONDS

```
Syntax
  integer = MICROSECONDS(absolute time)
  integer = MICROSECONDS(relative time)
```

MICROSECONDS returns the number of microseconds that the absolute or relative time represents. For an absolute time, this is the number of microseconds since 00:00:00 March 1, 1900, local time.

MIN

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  number = MIN(number1,number2)                                            │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

MIN returns as a value the smallest of its two arguments.

MIN_SUBSCRIPT

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  integer = MIN_SUBSCRIPT(array)                                           │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

MIN_SUBSCRIPT takes a fixed-bounds array as argument and returns the minimum value for the first subscript as integer value. If given an unbounded array, the value returned is the smallest subscript set so far.

QUOTE

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = QUOTE(string)                                                   │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

QUOTE returns as a value its argument string, surrounded by double-quotes, and with any interior double-quotes doubled. That is, QUOTE could be defined in terms of the DOUBLE built-in function as follows:

```
FUNCTION QUOTE(STRING)
EXIT '"'||DOUBLE(STRING,'"')||'"'
ENDFUNCTION
```

RELTIME

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  relative time = RELTIME(integer)                                         │
│  relative time = RELTIME(string)                                          │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

For integer arguments, RELTIME takes the value of its argument as the number of microseconds of relative time. If the argument is a string, it is assumed to be the character representation of a time interval. In both cases, the value returned is of type “relative time”. For example,

```
"1 week"
```

```
"3 days"
```

See Appendix G for a description of the legal forms.

REPLACE

Syntax

```
string = REPLACE(string1,string2,string3)
```

This is the same as the SNOBOL function of the same name. The second and third arguments are treated as parallel sets of characters and must be the same length. The output string is the same as "string1" with all occurrences of characters found in the second string replaced by characters in the third string. For example,

```
REPLACE("abcdef","bd","xy")
```

yields

```
"axcyef"
```

REVERSE

Syntax

```
string = REVERSE(string)
```

This is the same as the SNOBOL function of the same name. The output string is the same as the input string turned end for end.

RPAD

Syntax

```
string = RPAD(string,integer[,padstring])
```

RPAD requires two arguments: the first is the string to be padded and the second is the integer number of columns wanted for the final result string. An optional third argument is the string to be used for padding. If not given, the pad string defaults to a single blank. If the string is shorter than the number of columns wanted, it is padded on the right with the appropriate characters; otherwise the result string is the same as the argument string.

SENSE

```
Syntax
result = SENSE(string,integer)
result = SENSE(fdubptr,integer)
```

SENSE takes two arguments, the first of which is the same as for the CONTROL built-in function. The second is an integer giving the number of bytes of sense data wanted, up to a maximum of 255. The result is an array of two elements: the first gives the return code from the system subroutine that was called. If it is zero, the second element is the sense data. Note that this will be exactly the length of the second argument; the number of bytes that are meaningful depends on the system device-support routine ultimately called: it is supplied the length of the region but does not pass back the length it used, so it is not possible to set the correct length. If the return code is non-zero, the second element is the error message involved.

SIZE

```
Syntax
integer = SIZE(string)
```

SIZE takes a string as an argument and returns an integer specifying the number of characters in that string.

SUBSTITUTE

```
Syntax
string = SUBSTITUTE(string)
```

SUBSTITUTE takes a string as an argument and returns that string with all string substitutions, as flagged by "{...}" (or their equivalent), performed.

TRIM

```
Syntax
string = TRIM(string)
```

TRIM returns as a value the argument string, trimmed of all trailing blanks.

UPPERCASE

Syntax

```
string = UPPERCASE(string)
```

UPPERCASE takes a string as an argument and returns that string with all the lowercase letters converted to uppercase.

VARIABLE

Syntax

```
string = VARIABLE(string)
```

VARIABLE takes a string as argument and returns the null string if a user variable of that name does not exist, or returns its type otherwise. For example,

```
IF VARIABLE("I") DOESNT EXIST, DEFINE I=0
```

The function VARIABLE may be used to enquire about the elements of arrays. For example,

```
VARIABLE("A(I)")
```

In this case, the pieces of the expression must be defined or an error will occur. For the example, A must exist and be an array, I must exist, and I must have a value which is a valid subscript for the array A.

System Variables

A list of system variables may be obtained with

Syntax

```
DISPLAY SYSTEM_VARIABLES
```

In addition, the commands are actually variables, although not printed to reduce the size of the list and to reduce confusion.

System variables that are not names of built-in functions are:

BATCH

BATCH is TRUE or FALSE (Boolean) depending on if this is a batch or terminal session.

CLS_NAME

CLS_NAME is the name of the active CLS (command language subsystem). The value is a string.

CPU_TIME

CPU_TIME is the amount of CPU time consumed by the task since some unspecified time before the user signed on, expressed as a relative time.

CS_CODE

CS_CODE is the code set by last command or program executed. The value is an integer (zero is normal termination; non-zero is an error termination). See Appendix C for a description of the CSSET and CSGET subroutines.

CS_ORIGIN

CS_ORIGIN is the origin set by the last command or program executed (this specifiies where the CS_CODE came from). The value is an integer (–1 means unspecified; 1 means MTS set it). See Appendix C for a description of the CSSET and CSGET subroutines.

CS_SUMMARY

CS_SUMMARY is the summary of the status set by the last command or program executed. The value is an integer (0 is normal; 1 is warning; 2 is error). See Appendix C for a description of the CSSET and CSGET subroutines.

DATE

DATE is the current date, in same form as the MTS command DISPLAY DATE produces. The form is "Www Mmm DD/YY", where Www is the weekday, Mmm is the month, DD is the day of month, and YY is the last two digits of the year. For example, "Wed Dec 14/83".

FULL_SCREEN_POSSIBLE

FULL_SCREEN_POSSIBLE is True if the user is running at a terminal that is supported by the Full-Screen Support Routines, and False otherwise. This variable is evaluated only at the time the macro processor is initialized, e.g., when the $SET MACROS=ON command is given.

HOST_NAME

HOST_NAME is the name of the system, at the current installation, that the user is signed on to. For the University of Michigan, the two possibilities are UM and UB.

INSTALLATION

INSTALLATION is a string-valued code for the installation. For the University of Michigan, it is "UM".

INSTALLATION_CODE

INSTALLATION_CODE is an integer-valued code for the installation. For the University of Michigan, it is 1.

INSTALLATION_NAME

INSTALLATION_NAME is a string giving a name for the installation. For the University of Michigan, it is "MTS Ann Arbor".

LAST_SIGNON

LAST_SIGNON is the time/date of the last signon, expressed as an absolute time.

NONNULL

NONNULL is the null string with operator reversal.

NULL

NULL is the null string.

PROJECT

PROJECT is the 4-character project number for the current signon ID. The value is a string.

RATE

RATE is the current rate-class. The value is a string (“LOW”, “NORMAL”, “DEFERRED”, or “MINIMUM”).

RUNRC

RUNRC is the return code of last program execution completed. The return code is the value left in register 15 if the program terminated by returning. The value is an integer (if the program terminated by calling the subroutine SYSTEM, the value is set to 0; if it called ERROR, the value is set to 8; other terminations and suspensions of execution leave it set at –1).

SIGNONID

SIGNONID is the 4-character current signon ID. The value is a string.

TASKNBR

TASKNBR is the number of the current task. The value is an integer.

TERMTYPE

TERMTYPE is the 8-character terminal type (the contents of bytes 48–55 of a the data returned from a SENSE command to MSOURCE). The value is a string.

TIME

TIME is the current time, in same form as the MTS command DISPLAY TIME produces. The form is "HH:MM:SS ZZZ", where HH is hours, MM is minutes, SS is seconds, and ZZZ is the time-zone. For example, "21:09:07 EST".

These variables are all set automatically by the system.

How Expressions Are Formed

Expressions follow the standard rules of most programming languages for expressions. Parentheses may be used to force the order of evaluation, and when the parentheses are omitted, the order of evaluation depends on the precedence of the operators concerned.

Precedence of operators is as follows. The operators in a given entry have a higher precedence (i.e., they are done first) than the ones in the entries that follow it. All operators listed in the same entry have the same precedence. If the operators in an expression are of equal precedence, they are performed left to right.

(1) subscription
function call
substring
:

(2) – [unary]

(3) *
/

(4) +
– [binary]

(5) <
= [comparative]
>
<=
>=
¬=
IS
ISNT

(6) NOT

(7) AND

(8) OR

(9) | | [concatenation]

(10) = [substitution]
+=

```
-=
||=
```

Automatic Type Conversions in Expressions

Essentially what happens is that conversion is done only when necessary. Integer can always be converted to character. Integers that are not too large or too small can be converted to line-numbers. Line-numbers of integral value can be converted to integer. Line-number, integer and Boolean can go to character. Characters can be automatically converted to integers, line numbers, and Boolean, depending on the contents of the character string.

Character strings can be converted to arrays (if forced by subscription). The string is treated as a list denotation and converted to a one-dimensional fixed bounds array. Sublists become subarrays. If the string does not start with a left paren, it is converted to an array of one element. Arrays can be converted to strings if they are one-dimensional and fixed-bounds or unbounded, with lower-bound being 1. (These conversions are provided to allow handling lists and sublists, although they are generally available; see examples in the section, "Macros".)

Some operators, such as "+", require numeric operands and so the operands will be so converted. For the comparison operators, however, the comparison could be either numeric or character. However, the arguments to macros are always strings. For example,

```
ZMF  5  10
```

is the same as

```
ZMF "5" "10"
```

and if a comparison was done inside the macro, you would not expect the first argument to be greater than the second in the example above. So normally for comparison, if only one of the operands is a string but it can be successfully converted to a number, it will be so converted and the comparison will proceed numerically. If both operands are strings and both can be converted to numbers, both will be and the comparison will proceed numerically. Otherwise, a string comparison will be done. (Exception: a LOOP FOR iteration will force the iteration variable to be numeric, and if it contains a TO clause the terminating comparision will have the operands forced to numeric.) Note that in a macro the dummy variables that represent the arguments on a call are always strings, even if the calling argument is composed only of numeric characters.

CONTROL STRUCTURES

Labels

Labels are used to allow one to refer to a specific command. The label is defined by making it the only parameter on a LABEL command:

Syntax

```
LABEL label
```

Labels may only be defined in macros; they may not be defined in "open code."

Transfer of Control

A command is available to transfer unconditionally from one point of command interpretation to another. It is hoped that the other control structures offered will allow users to not use this except rarely. But if they want to, here it is:

Syntax

```
GOTO label
```

The label specified must belong to the same level (with regard to loop and conditional nestings) as the GOTO command. The GOTO command may be used only in macros; it may not be used in "open code."

Conditionals

The conditional exists in both the simple and compound form. The simple form is:

Syntax

```
IF bexp, command
```

where "bexp" is a Boolean expression. The "command" is executed if the expression is true.

The compound form is:

Syntax

```
IF bexp
          .
          .
          .
[ELSEIF bexp]
          .
          .
          .
[ELSE]
          .
          .
          .
ENDIF
```

In this case, if the first "bexp" is true, the first block of commands (up to the next ELSEIF, ELSE, or ENDIF at that nesting level) is executed and then control is transferred to following the corresponding

ENDIF. If the first "bexp" is false, then if the next "bexp" (in the first ELSEIF) is true, its block of commands is executed,... In other words, at most one block of commands is executed and it is the first one whose Boolean expression is true. The ELSE command is considered to be equivalent to ELSEIF TRUE.

Iteration

All of the forms of iteration can be used only in macro definitions, not in "open code." The iteration structure has the common basic form:

Syntax

```
LOOP how
        .
        .
        .
ENDLOOP
```

The "how" consists of zero or more optional clauses, each introduced by a keyword. The full prototype of the command is as follows:

Syntax

```
LOOP [FOR variable] [,]
     [FROM initial-value] [,]
     [BY increment-value] [,]
     [TO final-value] [,]
     [WHILE boolean-value] [,]
     [UNTIL boolean-value] [,]
     [OVER array-or-string-value]
```

None of the clauses have to be present. The only restriction is that if the OVER clause is given, the FROM, BY, and TO clauses may not, and vice versa. Multiple FROM/BY/TO clauses may be given and are processed sequentially in the order given. Multiple WHILE/UNTIL clauses may be given, and all are processed every time through the loop.

The expressions given for the FROM/BY/TO clauses are evaluated before entry to the loop and are never reevaluated. The expressions given for WHILE/UNTIL clauses are reevaluated at the front of the loop, every time through the loop.

If multiple termination conditions are given, the first one that is true will cause termination of the loop.

The commas between clauses are usually not required: the rule is that they may always be given, and are required preceding a keyword only if there is a user variable defined with the same name as the keyword.

Explanation of the various clauses is as follows:

(1) OVER expr

This form is usually used in a macro definition to iterate over the elements of a list provided as one of the parameters. If "expr" is a list of elements, then the number of times through the loop is the number of elements, and each time through the loop, the iteration variable gets set to the next element in the list.

For, example, if the user calls a macro

```
BARF (A,B,C+D) Z
```

and the macro was defined with two positional parameters, ARG1 and ARG2, and in the definition is found

```
LOOP FOR V OVER ARG1
     SET ARG2=ARG2+V
ENDLOOP
```

the loop will be processed three times, the first time with V inside the loop having the value A, the second time B, and the third time C+D.

(2) WHILE bexp

In this case the iteration continues as long as the Boolean expression "bexp" is true. If "bexp" is false when the LOOP statement is first encountered, then the commands in the scope of the loop will not be executed.

(3) UNTIL bexp

In this case the iteration continues until the Boolean expression "bexp" is true. If "bexp" is true when the LOOP statement is first encountered, then the commands in the scope of the loop will not be executed.

(4) FOR var

This clause specifies the name of the iteration variable to be used. If this clause is not present, an internal variable will be used. FOR is normally used along with FROM/BY/TO or OVER. A FOR clause is ignored if only WHILE or UNTIL clauses have been used.

(5) [FROM beg-expr] [BY incr] [TO end-expr]

This is a standard "increment variable" iteration. "beg-expr", "incr", and "end-expr" must all be expressions with numeric values. "beg-expr" is the value the iteration is to begin with and defaults to 1 if not given. "incr" indicates the increment to be used and defaults to 1 if not given. "end-expr" indicates the ending value and defaults to the maximum positive integer. More than one FROM/BY/TO set may be given on a single iteration. The second and subsequent sets are indicated by additional occurrences of the FROM clause. If multiple sets are given, they are processed sequentially: after the first is exhausted, the second is begun, and so on, and the iteration does not terminate until

after the last has ended (or some other termination condition has occurred). Thus,

```
LOOP FOR I FROM 1 TO 3, FROM 5 TO 7
```

will result in one iteration, with I taking on values of 1, 2, 3, 5, 6, and 7 on successive times through the loop.

This form of the iteration is a "shorthand" for writing the following:

```
   SET VAR var = beg-expr
   SET VAR termination = end-expr
   SET VAR increment = incr
   GOTO L1
LABEL L2
   SET VAR var=var+increment
LABEL L1
   IF var>termination, GOTO L3
        .
        .
        .
   GOTO L2
LABEL L3
```

(6) (no clauses at all given)

This is an infinite loop. Use of an EXITLOOP within the scope of this loop is therefore advisable.

Iteration Exits

In addition, the command

Syntax

```
NEXTLOOP
```

may be used to force the next iteration, and

Syntax

```
EXITLOOP
```

will cause transfer of command interpretation to the command following the ENDLOOP.

COMMENTS

Comments may be placed in macros and user-defined functions (and open-code) by placing them in a line that starts with ">*". As with all such "macro commands", the ">" must be the first character in the line and the first non-blank character following it must be the asterisk. Currently the ">" is

required, even in macro definitions: lines that begin with only an asterisk are just emitted by the macro processor. In MTS command mode they will be treated as MTS comments, so the net effect will be the same, albeit somewhat more expensive. However, if a program is reading the lines, it probably will not know what to do with them. Also, since user-defined functions can not currently emit lines, any such asterisk-only lines appearing in function definitions will cause a warning message for each such line encountered.

CONTINUATION LINES

The macro processor does not currently provide any way for a macro command to occupy more than one line. Please note that although certain of the examples in this document may have a macro command spread across more than one printed line, this is due to the formatting requirements of the document, and they should be viewed as really occupying only one line.

COMMAND MACROS

A command macro is a series of commands stored under a name and invoked using that name. Arguments may be provided, and other statements used to affect the flow of control.

Note: The current version implements *source* macros, but not yet general I/O stream macros. If $SET MACRO=ON has occurred, then all reads from *SOURCE* (or copies thereof) will go through the macro processor, unless the @NOMACRO modifier is used. (see Appendix A for a description of the @NOMACRO modifier). Appendix E has several complete examples of macros.

HOW TO INVOKE

A command macro invocation from MTS command mode looks like an MTS command invocation, that is, an optional "$" followed by the "command name", which is the macro name in this case followed by any associated arguments.

Similarly, a command macro invocation from EDIT command mode looks like an edit command. Note that in this case, putting a "$" on the front may cause different behavior, since the "$" means switch to MTS command mode before expanding the macro (see Appendix B for an example).

Arguments may be of the positional form, the keyword form, or the valueless keyword form. If any positional argument is to contain the characters blank, comma, equal-sign, or the "–>" digraph, then the argument must be surrounded with quotes. Likewise, if the value of a keyword argument is to contain any of those characters, then the value must be quoted.

If the last argument of a call is of the form

RC–>variable

this indicates that the return code from the macro is to be stored in the variable specified. The variable must be a simple variable (i.e., not an element of an array or structure). Specifying the RC–> in the call will suppress any call to the system subroutine CSSET to set the CS_SUMMARY and CS_CODE variables that would otherwise occur if an explicit return code was produced by the macro.

WHERE THE DEFINITION COMES FROM

Sources

Definitions can be provided in two ways:

(1) In-line

A macro definition may appear in the SOURCE stream read by the MTS command processor.

(2) From a file set up as a macro library

A library of command macro definitions, set up with a directory like an assembler macro library, may be set up using an FDname with the SET command:

```
SET VAR MACLIB(n)="FDname"
```

A macro library may be removed by using the null string with the SET command:

```
SET VAR MACLIB(n)=""
```

Search Order

The order of search is as implied above. To be precise, a “command name” provided by the user is inspected to see if

(1) it is a macro defined inline,

(2) it is defined in a macro library, looking first at that macro library attached at the highest MACLIB index, and then proceeding down toward zero,

(3) it is a macro command, or

(4) it is unrecognized, and hence emitted. If the line was being read in MTS command mode, then MTS looks to see if

   (a) it is an MTS command,
   (b) otherwise, it is an invalid MTS command.

If the @SYSTEM modifier is attached to the command name, then user definitions (1 and 2 above) are skipped. If the command name is the same as the name of the macro currently being expanded, then the @SYSTEM modifier is assumed implied unless the user explicitly overrides this with the @RECURSIVE modifier. This last condition applies only to macros used in “command” manner, not to macros used as user-defined functions. In functional use, it is assumed recursion is intended if a function calls itself.

To override the scoping due to a CLS (command language subsystem) or PKEY (program key) filter so that the macro definitions can be displayed from MTS command mode no matter how they are defined, one of the constructs

```
@CLS=xxx
```

or

```
@PKEY=xxx      (or @PKEY="xxx")
```

may be applied to the symbol (the "xxx" form for PKEY is required only if the pkey contains special characters). Note that the meaning of this override is “assume for this use of the symbol that the CLS (or pkey) is ...”. Thus,

```
>DISPLAY MACRO(MAYBE@CLS=EDIT)
```

when issued from MTS command mode will use the definition of MAYBE that is visible in EDIT mode. Note that this also allows the invocation of the “wrong” definition if desired, as for example entering

```
MAYBE@CLS=EDIT
```

in MTS command mode.

Scope

Macro definitions, like open-code user variables, are global in scope by default. For each macro name there is a global user variable of that name whose contents is the macro definition. A redefinition of a macro will automatically get rid of the previous definition; it is not necessary to FORGET it first.

Since it is a variable name, it can be used to a limited extent in expressions: it can be copied. Thus, abbreviations and synonyms may be provided by making copies under the names wanted.

For example, if there exists a macro called CREATE_OR_EMPTY:

```
MACRO CREATE_OR_EMPTY FILE
   . . .
ENDMACRO
```

then

```
DEFINE COE=CREATE_OR_EMPTY
```

will make COE a synonym for the longer name, and so use of either name will invoke the same macro.

DEFINING A MACRO

A definition is signalled with the MACRO command:

Syntax

```
MACRO name [parameters] [defnmod]
          .
          .
          .
ENDMACRO [name]
```

where “name” is the name of the macro. This creates a variable of type macro, and thus the same name cannot be used for both a macro and another variable. “parameters” is a list of items, which is discussed in the next section, and “defnmod” is a list of definitional modifiers (each beginning with an “@”), which is discussed later. Items of the list may be separated by either blanks or a comma. A definition is terminated with an ENDMACRO command:

Definitions may be nested, however, an inner macro is not defined until the outer macro has been called. If a definition is being processed and an ENDMACRO without a name or an ENDMACRO with the name of the definition being processed is encountered, the definition is terminated. An ENDMACRO with a different name is stored as part of the definition.

Macro Parameters

The parameters specified in the MACRO command are local variables to the macro that are set up by MTS when a call to that macro is made. They are set up to have values either specified by the caller in the argument list or default values specified in the MACRO command.

There are three types of parameters: positional, keyword, and valueless keyword.

Positional Parameters

Positional parameters appearing in the MACRO command parameter list require that the user of the macro substitute some value for each of them when the macro is invoked.

MTS will prompt the user for any that are missing, unless the MACRO command specifies the definitional modifier @NOPROMPT, in which case the missing items will have a null character string as the value. The definitional modifier may be overridden for specific parameters by attaching @PROMPT or @NOPROMPT to the parameter. The default case (which is to prompt) may be changed by use of the MACROPROMPT parameter on the SET command (values are ON or OFF).

The caller of the macro can give more positional parameters than the definition specifies. While these excess parameters may not be referred to by name (since they have none), they may be accessed by ordinal position, as in fact may the ones that do have names. A local one-dimensional array, POSITIONAL_PAR, is defined to allow this. {POSITIONAL_PAR(1)} will refer to the first parameter, {POSITIONAL_PAR(2)} is the second, etc. The local variable NBR_POSITIONAL_PAR is created and set to the number of positional parameters that actually occurred on the macro call. In addition to the parameters, the local variable PARSTRING is set to a value which is the entire set of parameters, starting with the first non-blank character of the first parameter.

The extraneous material on a macro call may be referred to as a single item with the name ETC. This includes excess positional parameters, as well as any excess keyword parameters if the @ETC definitional modifier was given.

The positional parameters are specified on the MACRO command by giving their names, as for example

```
MACRO GNURRS COME
```

which defines the macro GNURRS to have a single positional parameter named COME. When the macro is invoked, the value the user specified in the call is assigned as the value of the local variable COME.

Keyword Parameters

Keyword parameters are specified in the MACRO command as a name followed by an equal-sign "=", optionally followed by a value which is the default to be used if the user chooses not to specify this parameter when the macro is invoked. If this default value is to have break characters (e.g., blanks, commas) in it, it must be enclosed in quotes. For example,

```
MACRO GNURRS COME,FROM=VOODVORK
```

or

```
MACRO GNURRS COME,FROM="THE VOODVORK"
```

Valueless Keyword Parameters

A valueless keyword parameter is a word the user may or may not supply and all that is to be known is whether or not it is present. For example, in the MTS command

```
$RUN OBJPROG MAP
```

the word MAP is such a parameter.

These parameters are specified in the MACRO command by giving their name enclosed in quotes, to indicate that the only thing the user can specify is that string or its negation. When the macro is invoked, if the user specified the keyword, the value of that local variable is set to PRESENT. If the user specified the keyword prefixed by "NO", "¬", or "–", the value is set to NEGATED. If the user did not mention it, the value of the local variable is set to ABSENT. For example, extending the example given above with the valueless keyword OUT, we have

```
MACRO GNURRS COME,FROM="THE VOODVORK","OUT"
```

and when the macro is invoked, the local variable OUT is set to PRESENT, NEGATED, or ABSENT, depending on whether or not the user said OUT, NOOUT, or ¬OUT or –OUT, or did not mention it in the invocation. The test for matching is performed in a case-independent manner, i.e., an all-uppercase copy of the definition is compared with an all-uppercase copy of the parameter.

If the valueless keywords are to be matched up only after all positional parameters are satisfied, then the positional parameters should come first on the MACRO command. If the valueless keywords are to take precedence over the positional parameters, then the valueless keywords should be first on the MACRO command. The processing is done left to right on the MACRO command specification, thus, for example, given

```
MACRO GLORP A,"TO",B
```

the first parameter the user specifies will match the positional parameter A. The second parameter the user enters will either match the TO (if that is what was entered) or else match positional parameter B.

Detailed Syntax of the Macro Prototype

Syntax

```
verb<bl>name[<di>par[<dm>par...]<dt>] [<bl>@defnmod[<bl>@defnmod...]]
```

where

verb — is either MACRO or FUNCTION

<bl> — is one or more blanks

name — is the name of function or macro being defined

<di> is the initial delimiter of either one or more blanks, or else a left parenthesis followed by zero or more blanks

par represents zero or more parameters, each of which is of one of the following forms:

(1) identifier
(2) string whose value is identifier
(3) identifier=characters
(4) identifier=string

where "characters" means zero or more consecutive characters, terminated by a blank, comma, or right parenthesis, and "string" means zero or more consecutive characters surrounded by either primes before and after or else double quotes before and after (whichever character is used as delimiter is represented within the string as two consecutive ones).

Form 1 represents a positional parameter, form 2 a valueless keyword parameter, and forms 3 and 4 represent keyword parameters. For keyword parameters, form 4 may always be used instead of form 3. Form 4 must be used instead of 3 if the character string contains blanks, commas, or right parentheses.

<dm> is medial delimiter of either one or more blanks, or else zero or more blanks followed by a comma followed by zero or more blanks

<dt> is trailing delimiter. If initial delimiter was a left parenthesis followed by zero or more blanks, then this must be a right parenthesis followed by zero or more blanks, otherwise it must be zero or more blanks.

defnmod is zero or more definitional modifiers: ETC, PROMPT, NOPROMPT, CLS=, PKEY=, or RAW.

INVOCATION OF A MACRO

When a macro call is made, all variables named in the definition list on the MACRO command are defined as local variables for the macro call, and are given initial values from the macro call parameters as follows:

The parameters in the macro call are assumed delimited by blanks or zero-level commas. If a parameter's value is to contain blanks, zero-level commas, equal-signs, or the "–>" digraph, then the value must be enclosed in string delimiters. These delimiters are removed before the value is assigned to the appropriate variable. A null string value may be supplied for a positional parameter by either putting an explicitly delimited null string (e.g., "") in the parameter list or else by providing multiple commas as delimiters.

The macro calling parameter list is processed left to right. Positional parameters are matched to ones on the defined list. Missing positional parameters are given null values or are prompted for, as requested. Excess positional parameters are ignored. Keyword parameters are assigned values found in the calling list, or if not present, the default from the definition is used. Keyword parameters present in the call but not in the definition are ignored (but added to ETC) if the @ETC definitional modifier was used and are illegal otherwise. Missing valueless keyword parameters are assigned the value ABSENT.

In addition, all positional parameters present on the call are defined as elements of the local variable one-dimensional array POSITIONAL_PAR under their ordinal numbers, so {POSITIONAL_PAR(1)} represents the first, etc.

The variable NBR_POSITIONAL_PAR is assigned the number of positional parameters that actually occurred on the call.

The variable ETC represents all calling parameters that were not assigned to formal parameters. This includes both excess positional parameters and unknown keyword parameters, the latter only if the @ETC definitional modifier was used in defining the macro (the default case is that unknown keyword parameters are illegal.).

The variable PARSTRING is set up to have as a value the entire unparsed parameter string, starting with the first character of the first parameter.

The variable MACRO_NAME (or FUNCTION_NAME, if a function is being defined) is assigned as value the name of the macro (or function) being defined.

If the definitional modifier @RAW is specified on the definition of a parameterless macro, no processing of the call into separate parameters will occur. In addition, for top-level calls (i.e., calls not from within another macro), no substitution (as specified with braces) will occur. None of the above-mentioned variables will be set up except for PARSTRING, which will be set to the untouched remainder of the input line starting with the first non-blank character following the macro name. This is the only way that arbitrary text can be passed on the call.

As an example of the interactions of name and value on calls, consider the following two macro definitions

```
MACRO CON1 A B          MACRO CON2 A B
WRITE {A}||{B}          WRITE {A||B}
OUTPT {A}||{B}          OUTPT {A||B}
ENDMACRO                ENDMACRO
```

where, in each case, OUTPT is a macro that merely writes out its argument:

```
MACRO OUTPT A
WRITE A
ENDMACRO
```

Now set up a few preliminary variables

```
DEFINE Z = 5
DEFINE X = 2
DEFINE ZX = 3
```

and call each macro with the same set of arguments, and compare results. First use constants

```
CON1  5  2              CON2  5  2
52                      52
5||2                    52
```

and the second time use variables whose values are those same constants

```
CON1  Z  X              CON2  Z  X
52                      3
```

```
Z||X                    ZX
```

LISTS AND SUBLISTS

Macro parameters (or any variable whose value is a string) that have the "traditional list" form may be processed by treating the parameter as an array and subscripting it to obtain the desired element. If, for example, parameter ZMF has value

```
(A,B,C,(D,E))
```

then ZMF(1) will extract A, ZMF(4) will extract (D,E), and ZMF(4)(2) will extract E.

The built-in function MAX_SUBSCRIPT may be used to obtain the number of elements. If a parameter has the form "A", this is assumed the same as "(A)" when being subscripted (i.e., any parameter can be subscripted, but if it is not a list then it only has one element). Note that if

```
DEFINE Z = "ABC"
```

then Z(1) has the value ABC,
and Z(2) does not exist.

The substring notation must be used if the second character of Z is desired:

```
Z(2|1)
```

or

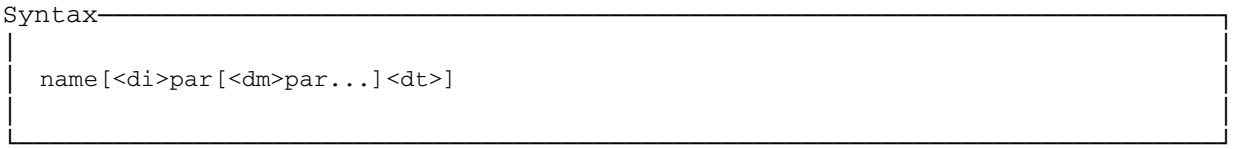
```
Z(2...2)
```

Now, however, if

```
DEFINE Z = "(A,B,C)"
```

then Z(2) has the value B,
whereas Z(2|1) has the value A,
and Z(2...2) has the value A.

Detailed Syntax of the Macro Call



where

name is name of function or macro being called

<di> is initial delimiter of either one or more blanks, or else a left parenthesis followed by zero or more blanks

par represents zero or more parameters, each of which is of one of the following forms:

(1) identifier=characters
(2) identifier=string
(3) identifier
(4) NOidentifier
(5) ¬identifier
(6) –identifier
(7) characters
(8) string
(9) RC–>identifier

where "characters" means zero or more consecutive characters, terminated by a blank, comma, or right parenthesis, and "string" means zero or more consecutive characters surrounded by either primes before and after or else double quotes before and after (whichever character is used as delimiter is represented within the string as two consecutive ones).

Forms 1 and 2 are for keyword parameters. With keyword parameters, form 2 may always be used instead of form 1. Form 2 must be used instead of 1 if the character string contains blanks, commas, or right parentheses. Forms 3 through 6 are for valueless keyword parameters. Using form 3 will cause the appropriate variable to have value PRESENT; using forms 4, 5, or 6 will cause a value of NEGATED. Forms 7 and 8 are for positional parameters. Form 8 may always be used; form 8 must be used if the characters contains one or more blanks, commas, or equal signs, or if you want to prevent it from matching a valueless keyword parameter of that name. Form 9 may only appear as the last "parameter" and indicates return code capture.

<dm> is medial delimiter of either one or more blanks, or else zero or more blanks followed by a comma followed by zero or more blanks

<dt> is trailing delimiter. If initial delimiter was a left parenthesis followed by zero or more blanks, then this must be a right parenthesis followed by zero or more blanks, otherwise it must be zero or more blanks.

When macros or user-defined functions are called as functions, the call is from within an expression and must use the parenthesized list (with comma separators) form. When macros or user-defined functions are called as macros, either form may be used. Note, however, that with a macro call in parenthesized form, the following two are not the same

```
Z(A,B)
Z  (A,B)
```

Although both invoke the macro Z, in the first case it will be called with two arguments, A and B, whereas in the second case it will be called with one argument, "(A,B)".

EXITING A MACRO OR USER-DEFINED FUNCTION

Exiting a Macro

The EXIT command is used to exit a macro:

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  EXIT [CODE=exp]                                                          │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

The “exp” specifies the return code. The expression must result in an integer or be convertable to one. If not specified via the CODE keyword, an implicit return code of zero is supplied.

Exiting a User-Defined Function

The EXIT command is used to exit a user-defined function:

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  EXIT value[,CODE=exp]                                                    │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

The “exp” specifies the return code. The expression must result in an integer or be convertable to one. If not specified via the CODE keyword, an implicit return code of zero is supplied.

The “value” is an expression which specifies the value to be returned from a user-defined function. If it is not specified, an “undefined value” is used.

Common to Both

Falling out of the bottom of a macro or user-defined function definition will also cause an automatic return, with an implicit return code of zero, and (if a function) a value of “undefined value”.

If a return code is explicitly given on the EXIT command, and the macro call did not specify (via “RC–>variable”) the capture of the return code, then the system variables CS_SUMMARY and CS_CODE will be set: CS_CODE will be set to the return code value and CS_SUMMARY will be set to 2 (unless the value is zero, in which case CS_SUMMARY also will be set to zero).

CONTROL AND MONITORING

Monitoring the progress of a macro can be done by means of options on the macro SET command or by using the @CHECK modifier when invoking the macro.

MACROECHO Option

Syntax

```
SET MACROECHO={OFF|ERROR|ON|ALL}
```

where	OFF	means never echo,
	ERROR	means echo any command that produces RC>4, as well as its error message,
	ON	means echo all non-control-flow (existing) commands,
	ALL	means echo everything

MACROTRACE Option

Syntax

```
SET MACROTRACE={ON|OFF}
```

If ON, all lines generated by a macro call and all lines that have symbolic substitution in them are printed (on SINK), prefixed with a header of the form:

*<macro-name>(<macro-linenumber>)<flag> <flow>

where “flag” is

“g” for macro generated lines,
“s” for lines that have been substituted into,

and “flow” is

“XXX” for lines processed while skipping is in effect,
“**...**” for derived lines, and
null otherwise.

The @CHECK Option

If the @CHECK modifier is appended to the macro name when a macro is invoked, then any lines that this macro would have emitted are instead printed on SINK. The lines are prefixed by “*C_”. If a macro is invoked in check mode, then any macros it calls are also processed in check mode.

```
COMPILE@CHECK MYPROG LISTING
```

INPUT AND OUTPUT FROM MACROS

There are two commands, READ and WRITE, for doing I/O from macros. Note that currently all expressions on the READ and WRITE commands (only) must have no blanks in them (except inside of string constants).

Input

Input is done using the READ command, which has the form:

```
Syntax
READ [invar]
     [FROM {fdname|FDUB=var}]
     [{MODS=mm|MODIFIERS=mm|@mm}]
     [{LNR|LINENUMBER}{=lexp|->lvar}]
     [PREFIX=sexp|*]
     [FDUB->fvar]
     [ATTN->bvar]
     [EOF->bvar]
     [ERR->bvar [ERRMSG->svar]]
     [ERRRC->ivar]
     [[AND] RELEASE]
```

In general, items specified with “name=value” describe values that are fetched, and those specified with “name–>var” describe values stored in the appropriate variables.

“invar” is the variable to be read into. It may be a simple variable name or an array element or a structure field. If it is not present then no input is done. In this case either the RELEASE or FDUB clauses are required (otherwise it would be an expensive nothing, which always looks suspicious).

The FROM clause specifies where the input is coming from. If omitted, the input is read from SOURCE. If present, it specifies either a file name or else a variable containing a FDUB-pointer (which was obtained via a FDUB clause earlier).

The MODIFIERS clause is used to force the read to be done with certain I/O modifiers (for example, INDEXED, i.e., read a particular line number). The modifier name may be given as the value of the keyword MODIFIERS (or MODS) or may be preceded by an at-sign “@” and used as a separate (positional) parameter. More than one modifier may be specified by using multiple MODIFIERS keywords (or equivalent), each specifying a different modifier. The complete list of I/O modifiers available is as follows:

I (or INDEXED)
BIN (or BINARY)
SP (or SPECIAL)
UC (or CASECONV)
NOCC ¬CC
IC

¬IC
TRIM
¬TRIM
ENDFILE
¬ENDFILE
LOG
¬LOG

The LINENUMBER clause either supplies an expression “lexp” to be used as the line number for the indexed read (if MODIFIERS specified indexed), or else provides a variable that will be set to the line number of the incoming line from a sequential read.

The PREFIX clause sets the prefix to be printed prefixing the read operation (the prefix specified will be truncated to 32 characters if it is longer). “sexp” is a string constant or an expression that can be converted to a string. If the PREFIX clause is not given, PREFIX="?" is assumed. If the prefix clause is given as PREFIX=*, then the prefix is not changed from whatever it was before.

The FDUB–> clause saves the FDUB pointer used for the I/O operation in the specified variable, so it can be used in the FDUB clause in subsequent I/O commands.

The ATTN, EOF, and ERR clauses specify Boolean variables that are set to TRUE if the respective condition (attention interrupt, end-of-file, or error return) occurred on the read operation, and are set to FALSE otherwise. If one of these conditions occurs and the respective clause was *not* present in the command, then an error message is printed and macro processing is aborted.

If both the ERR and ERRMSG clauses are present, then if an error occurs the variable specified by the ERR clause is set to TRUE and the variable specified by the ERRMSG clause is set to contain the error message, if one was available.

The ERRRC clause specifies a variable whose values is set to the return code resulting from the I/O operation.

The RELEASE clause is used to explicitly release the FDUB acquired for the I/O operation for both the “fdname” and FDUB cases. Normally, when the first read or write is made to a given FDname, a FDUB is acquired at that point, and retained for subsequent I/O operations using the same FDname. This FDUB is normally released only when back at “zero-level” of macro invocation. The FDUB may be explicitly released after the I/O operation by appending the AND RELEASE clause. For example,

```
READ DIRLINE FROM ETC:MSGFILE MODS=INDEXED LNR=IX AND RELEASE
```

and an example using FDUB variables:

```
macro copyfile fn
define z
define end
define l
read from {fn} fdub->z
loop
  read l from fdub=z eof->end
  if end, exitloop
  write l
endloop
read from fdub=z  release
endmacro
```

Output

Output is done using the WRITE command, which has the form:

Syntax

```
WRITE [exprs]
      [ON {fdname|FDUB=var}]
      [{MODS=mm|MODIFIERS=mm|@mm}]
      [{LNR|LINENUMBER}=lexp]
      [PREFIX=sexp|*]
      [FDUB->fvar]
      [ATTN->bvar]
      [ERR->bvar [ERRMSG->svar]]
      [ERRRC->ivar]
      [[AND] RELEASE]
```

In general, items specified with “name=value” describe values that are fetched, and those specified with “name–>var” describe values stored in the appropriate variables.

“exprs” is the expression to be written out. If it is not present then no output is done. In this case either the RELEASE or FDUB clauses are required (otherwise it would be an expensive nothing, which always looks suspicious).

The ON clause specifies where the output is going to. If omitted, the output is written on SINK. If present, it specifies either a file name or else a variable containing a FDUB-pointer (which was obtained via a FDUB clause earlier).

The MODIFIERS clause is used to force the write to be done with certain I/O modifiers (for example, INDEXED, i.e., write a particular line number). The modifier name may be given as the value of the keyword MODIFIERS (or MODS) or may be preceded by an at-sign “@” and used as a separate (positional) parameter. More than one modifier may be specified by using multiple MODIFIERS keywords (or equivalent), each specifying a different modifier. The complete list of I/O modifiers available is as follows:

I (or INDEXED)
BIN (or BINARY)
SP (or SPECIAL)
UC (or CASECONV)
NOCC
¬CC
IC
¬IC
TRIM
¬TRIM
ENDFILE
¬ENDFILE
LOG
¬LOG

The LINENUMBER clause supplies an expression (“lexp”) to be used as the line number for the indexed write (if MODIFIERS specified indexed).

The PREFIX clause sets the prefix to be printed prefixing the write operation. (The current version will truncate the prefix specified to 32 characters if it is longer.) “sexp” is a string constant or an expression that can be converted to a string. If the PREFIX clause is not given, PREFIX=" " is assumed. If the prefix clause is given as PREFIX=* then the prefix is not changed from whatever it was before.

The FDUB clause saves the FDUB pointer used for the I/O operation in the specified variable, so it can be used in the FDUB clause in subsequent I/O commands.

The ATTN and ERR clauses specify Boolean variables that are set to TRUE if an attention interrupt or error return (respectively) occurred on the write operation, and are set to FALSE otherwise. If an attention interrupt occurs and the ATTN clause was *not* present or an error return occurs and the ERR clause was *not* present, then an error message is printed and macro processing is aborted.

If both the ERR and ERRMSG clauses are present then if an error occurs, the variable specified by the ERR clause is set to TRUE and the variable specified by the ERRMSG clause is set to contain the error message, if one was available.

The ERRRC clause specifies a variable whose values is set to the return code resulting from the I/O operation.

The RELEASE clause is used to explicitly release the FDUB acquired for the I/O operation for both the “fdname” and FDUB cases. Normally, when the first read or write is made to a given FDname, a FDUB is acquired at that point, and retained for subsequent I/O operations using the same FDname. This FDUB is normally released only when back at “zero-level” of macro invocation. The FDUB may be explicitly released after the I/O operation by appending the AND RELEASE clause.

```
WRITE " MZB019I Questionable use of question mark?"
```

USER-DEFINED FUNCTIONS

A “user-defined function” is a macro which returns a value instead of emitting lines as output. Its definition may be exactly like the definition of macros given above, with the difference being the value returned, which is specified by giving an expression on the EXIT command. Preferably, the prototype line of the definition should be expressed with parentheses and commas in the traditional functional manner, with the definitional commands being FUNCTION and ENDFUNCTION instead of MACRO and ENDMACRO.

Syntax

```
FUNCTION name([parameters]) [defnmod]
          .
          .
          .
ENDFUNCTION [name]
```

Calling a user-defined function is exactly like calling a built-in function: the name of the function followed by a parenthesized list of expressions which are the arguments is used where the value of the function is wanted in an expression.

See Appendix E for several complete examples of user-defined functions. For example, the following macro computes (the hard way) the sum of the two arguments:

```
FUNCTION ADD(A,B)
EXIT A+B
ENDFUNCTION
```

and a sample use of this is

```
IF ADD(ARG1,ARG2) ISNT 0, BARF
```

As another example, the infamous Ackermann's function is

```
FUNCTION ACKER(M,N)
IF M=0
   EXIT N+1
ELSEIF N=0
   EXIT ACKER(M-1,1)
ELSE
   EXIT ACKER(M-1,ACKER(M,N-1))
ENDIF
ENDFUNCTION
```

with sample calls of

```
WRITE ACKER(0,0)
WRITE ACKER(1,0)
WRITE ACKER(1,1)
```

User-defined functions and macros are really the same things and can be used interchangeably. The only major difference is that using a macro or function as a function returns a value whereas using a macro or function as a macro emits lines (usually). However, the invocation of a function occurs only in an expression, wherein identifiers in the argument list are treated as names, whereas in a macro invocation, all arguments are strings.

If we have

```
DEFINE Z = "ZZ"
DEFINE X = "XX"
```

and now define an ambidextrous macro

```
MACRO CONC A B
WRITE A||B
EXIT A||B
ENDMACRO
```

Now if we call it as a macro

```
CONC(Z,X)
```

we get

```
ZX
```

but if we use it as a user-defined function

```
WRITE CONC(Z,X)
```

we get

```
ZZXX
ZZXX
```

MACRO LIBRARIES

A macro library is a file containing several macro definitions and a directory, in a specific format. Providing the macros in this form rather than $SOURCEing to a file containing their definitions allows the system to defer reading in the definition of a macro until it is needed. This is particularly useful if the library contains a large number of macro definitions and only a few (or maybe none) are needed during a session.

The preferred method of using the macro processor is to set up a macro library file and then place in the sigfile the MTS command

```
$SET MACRO=ON
```

followed by the macro command to establish this file as the macro library (see below). This way all of the macros will be set up each time the user signs on.

Using a Macro Library

A file is established as a macro library by setting its name as an element in the system array MACLIB:

Syntax

```
SET VAR MACLIB(n)="filename"
```

where “n” is an integer (use 1 if only one macro library is being established).

When the SET VAR MACLIB command is entered, the directory at the front of the specified file is read. Every symbol in the directory is defined as a macro whose definition is not present, except that if any of the symbols is already defined, the current definition is not changed. When (or if) one of these symbols whose definition is not present is referred to, the definition is read in at that time.

Note that because of the structure of a macro library, a given name can occur only once. Hence to have two different macros with the same names but different filters they must be in different macro libraries.

If a SET VAR MACLIB(n)="" command is later entered, all symbols that were defined by reading the macro library directory earlier are now forgotten, except that any symbol whose value has been changed between then and now is not removed.

Multiple Libraries

Multiple macro libraries can be used simultaneously. The global system variable MACLIB (an array) keeps track of them. If a symbol is not found in the global user symbol table, then the library set up under the highest subscript of MACLIB is searched next, then the next-highest, and so on down. The element MACLIB(0) is pre-assigned to *CMDMACLIB, which is the system macro library.

To turn off a macro library at any index, assign a null string to MACLIB. For example,

```
SET VAR MACLIB(1)=""
```

Note that DISPLAY MACROS displays only the in-line macros, which go in the global user symbol table. To find out the names of the macros at a given level, MACLIB may be displayed either by name or index, for example

```
DISPLAY MACLIB(1)
```

or

```
DISPLAY MACLIB("zilch")
```

To find out what macro libraries are active, the contents of the array MACLIB may be displayed:

```
DISPLAY ARRAY(MACLIB)
```

Form of a Macro Library

The form of a macro library that is specified with a SET VAR MACLIB(n)="FDname" command is as follows:

The directory starts at line 1. Each line has the name of a macro and the line number at which its definition starts, separated by one or more blanks. The line number must be an integer. The directory is terminated with an end-of-file, a line consisting of "/END", or the terminator "00000000". The remainder of the file contains the definitions, each beginning with MACRO or FUNCTION and ending with ENDMACRO or ENDFUNCTION. The line the directory entry points to must be the MACRO line.

Constructing a Macro Library

A macro library can be constructed by hand, using the editor. For larger libraries, use of the LIBGEN utility program is recommended. A description of LIBGEN is given in Appendix D.

System Macro Library

One macro library is always supplied by default, and is attached to MACLIB at subscript 0 so it will be searched last. This macro library is in the file *CMDMACLIB. It currently contains only one macro, MACROLIB, which is used to attach other macro libraries. This macro is more convenient than the notation in given earlier because it keeps track of the indices and requires less typing; it offers no more power. Thus, instead of typing in

```
SET VAR MACLIB(1)="ZILCH"
```

one can just say

```
MACROLIB ZILCH
```

In addition,

```
MACROLIB LIST
```

will produce a list of what libraries are currently attached.

EDITING MACROS

The body of a macro definition is stored in an editor virtual file, and hence the editor can be used to modify it by issuing the editor command (not the MTS command):

```
EDIT MACRO x
```

You cannot edit a macro that is being expanded, nor can you expand (or forget) a macro that is being edited. You can change only the text of the macro, not the prototype.

A macro definition in a macro library can be edited in that macro library file up to the time at which that macro is first expanded. After that, a new SET VAR MACLIB(n) must be issued to force the new definition to be read in on the next invocation.

SPECIAL CASES

Emitting Lines

If a line present in a macro definition is not recognized by the macro processor as a macro command, it is emitted. So any lines you want the macro to generate can usually be just placed in the definition. If, however, you want to emit a line that would normally be recognized by the macro processor as something it should handle, then you can use the EMIT macro command.

Syntax

```
EMIT expression
```

where “expression” is what you want emitted. Thus, to have a macro produce an MTS command “SET ENDFILE=NEVER”, the following could occur in the definition

```
EMIT "SET ENDFILE=NEVER"
```

Generating Unmentionables

There are several things one might like a macro to generate that are "unmentionable" in the sense that they cannot be written. These include generating null lines, end of files, and attention interrupts. These can be produced using the GENERATE command.

Syntax

```
GENERATE ENDFILE
```

will cause the macro processor to cause MTS to generate a real end-of-file (return code of 4) for the next "input line" it is asked to fetch from *SOURCE*.

Syntax

```
GENERATE NULL
```

will cause a null line (a line of zero length) to be emitted

Syntax

```
GENERATE ATTN
```

will cause MTS to behave as though an attention interrupt had occurred.

FILTERS

A filter is a restriction on a macro definition which provides further limitations on when that symbol (and its definition) will be recognized. A filter may require that a given CLS is or is not in control, or that the current program key is or is not a specified pkey. This restriction is specified by a definitional modifier on the macro prototype line, in one of the forms:

@CLS=cname
@CLS¬=cname
@PKEY=pkey
@PKEY¬=pkey

where "cname" is a CLS name, and "pkey" is a program key. If the program key contains non-alphanumeric characters, then its value should be quoted, as in "@PKEY="ABCD:PROG"'"

For example, suppose we want to have a macro PRINT that is to be used in MTS command mode, but when we are in the editor we want PRINT to mean the editor's PRINT command. We could define

```
MACRO PRINT Z @CLS=MTS
  . . .
```

```
ENDMACRO
```

which will define a macro named PRINT, but this symbol will be recognized only in MTS command mode. We could also define it as

```
MACRO PRINT Z @CLS¬=EDIT
  . . .
ENDMACRO
```

and then it would be defined everywhere except in edit mode.

Note that this filtering spplies to all uses of the symbol, not just macro invocation. Using the above example, if you entered a "DISPLAY MACRO(PRINT)" command while in the editor, it would tell you that symbol PRINT does not exist, whereas if you issued the same command from MTS command mode you would get a listing of the macro.

To override this, the symbol may be qualified with "@CLS=cname" or "@PKEY=pkey", which may be interpreted meaning "for this use of the symbol, assume the CLS is 'cname' (or the program key is 'pkey') instead of what it really is.". Thus a "DISPLAY MACRO(PRINT@CLS=MTS)" command will list the macro even if issued from editor mode.

Multiple Definitions

If you want to have, say, a COAGULATE macro which has one definition in MTS command mode and another in EDIT mode, be aware of the following:

(1) Macros defined inline are global user symbols, and there can be only one definition for a given name. If you try to enter both definitions inline, the second will just quietly replace the first.

(2) In each macro library, a given symbol can be defined at most once.

Hence to have two different definitions active at once, either one may be defined inline and the other in a macro library, or (more likely) the two definitions must be put in two different macro libraries.

Editing

If multiple definitions of a given macro are active, then all the definitions will be in the same virtual file, but each will start at a line number 10000 higher than the previous.

APPENDIX A
MACRO AND MACRO_FLAG_REQUIRED MODIFIERS

The macro processor in MTS functions as an I/O preprocessor on lines being read by MTS itself, commands, utilities, and user programs. Whether the macro processor gets called and what it expects on lines it sees is controlled by two I/O modifiers.

@MACRO and @NOMACRO control whether the processor is called at all to interpret lines for invocations or macro commands. It has no effect on the generation of lines: once a macro is invoked, the lines are generated for successive reads whether or not those reads have @MACRO or @NOMACRO applied.

@MACRO_FLAG_REQUIRED (or @MFR) and @NOMACRO_FLAG_REQUIRED (or @NOMFR) control whether the macro flag character ">" is required on lines to be inspected for macro invocations. It does not control lines to be inspected for macro commands; these always require the flag character.

The default for the *SOURCE* FDUB and all of its copies is @MACRO@MFR, but when MTS or any of the CLS's command processors read a command, they read @NOMFR. So to get the same effect when running a program, one could say "SCARDS=*SOURCE*@NOMFR" on the RUN command.

These modifiers may be applied to FDnames, as in

```
RUN PROGR SCARDS=*SOURCE*@NOMACRO
```

These modifiers may also be specified by setting the appropriate bits on the call to the READ or SCARDS subroutines. The bits are in the second word of modifiers, so the first word of modifier bits must have X'01000000' set to indicate that there is a second word of bits following the first word. The relevant bits in the second word are as follows:

Bit 29	NOMACRO	X'00000004'
Bit 28	MACRO	X'00000008'
Bit 27	NOMFR	X'00000010'
Bit 26	MFR	X'00000020'

APPENDIX B
INTERACTIONS OF FLAG CHARACTERS AND MODES

Consider the following instructive (albeit contrived) example. We have defined a macro D as follows:

```
MACRO D N
COPY {N}
ENDMACRO
```

This is the way it would appear in a macro library. If we were entering this in-line, we would have to put macro flag characters ">" on the MACRO and ENDMACRO commands.

Now let us enter the following four lines from MTS command mode

```
(1)  D 17
(2)  $D 17
(3)  >D 17
(4)  $>D 17
```

All four of these will cause same thing to happen: the macro D will be invoked, and the line "COPY 17" will be emitted and interpreted as an MTS COPY command to copy the file named "17" to *SINK*.

Secondly, let us enter "EDIT –T" and then enter the same four lines again. Again, in all four examples the macro D is invoked. But in cases (1) and (3) it is invoked from EDIT command mode, and the line "COPY 17" that is emitted will be interpreted by the editor as an editor copy command to copy line 17 to the current line, whereas in cases (2) and (4), the "$" causes the editor to pass the line to MTS to be interpreted, and so it is invoked in MTS command mode, and the "COPY 17" that is emitted is treated as the MTS COPY command to copy the file named "17" to *SINK*.

Thirdly, let us return to MTS command mode, enter "COPY *SOURCE* *SINK*", and then enter the same four lines again. Because the COPY command is reading with the default modifiers of @MACRO@MFR, the first, second, and fourth lines will be copied without change, but the third line will cause the macro to be expanded, so the following will appear:

```
D 17
$D 17
COPY 17
$>D 17
```

Finally, enter ">FORGET D" to get rid of our macro definition, and then enter the same four lines again. Now the first and second lines will be interpreted as (illegal) MTS DISPLAY commands and the third and fourth lines will be interpreted as (illegal) macro DISPLAY commands.

APPENDIX C
CSGET AND CSSET SUBROUTINE DESCRIPTIONS

CSSET

This routine is used to set a current command status. The command status currently consists of three parts: summary, code, and origin. Once set, the command status is accessible via subroutine CSGET, the macro processor, or the COMMAND subroutine.

All parameters are fullword integers.

Parameters (S-type):

Summary: Error/status summary.

0 (CS_NORMAL)	Normal command status
1 (CS_WARNING)	Warning or informational message
2 (CS_ERROR)	Command error

Code: This gives the status in more detail. Currently the only assigned numbers are for global-level errors issued by MTS, and for error conditions flagged by the MOUNT command. Other commands use –1 (unknown) in the interim.

User programs calling CSSET may select their own set of codes; the values must be non-negative.

Codes used by MTS are:

1 (CS_ATTN)	Untrapped attention interrupt
2 (CS_PGNT)	Untrapped program interrupt
3 (CS_SVC)	SVC error
4 (CS_SVCEXIT)	SVC EXIT happened
5 (CS_TIMER)	Untrapped timer interrupt
100 (CS_CMD_SYNTAX)	Command syntax error
101 (CS_RUNONLY)	Not legal with run-only program
102 (CS_LSSMODE)	Not legal in LSS mode
103 (CS_STAFF_CMD)	Only legal from Computing Center staff ID
104 (CS_PRIV_CMD)	Only legal from privileged ID
105 (CS_CLS_LOADING_ERROR)	Error occurred trying to load CLS
106 (CS_CLS_ERROR_RETURN)	Error return from CLS
200 (CS_GETSPACE)	Unable to obtain space
201 (CS_CANCEL)	User responded to prompt with CANCEL

Origin: Originator of the error/status information (optional).

If this parameter is omitted on a call to CSSET, the originator is set to –1 (indicating an undefined or undeclared state). Currently, only MTS sets the originator code, and it sets it to 1. In the future, each CLS and many public programs will have their own unique originator codes. For the present, user programs should either omit this parameter or else set it to –1 when calling CSSET.

The following constraints on parameter values exist: "Summary" can be zero if and only if "Code" is zero too. "Summary" must be one of (0,1,2). "Code" must be >= –1. "Origin" must be >= –1.

Eventually there may be other parameters, so the VL-bit is checked for to make sure more parameters can be added later.

Return codes:

0 – Status information set successfully.
4 – Illegal summary/code values; bad parameter list; no VL bit; etc.

CSGET

This routine is used to retrieve the current command status (i.e., whatever was set on the last call to CSSET).

This command status information is useful primarily in two situations:

(1) User programs that have called the COMMAND subroutine may call CSGET to determine whether the MTS command executed properly. The three values obtainable by calling CSGET are also available by specifying additional parameters on the COMMAND subroutine call.

(2) MTS command macros may be constructed to determine whether an MTS command executed properly. The three values that the CSGET subroutine returns are available within the macro processor as the system variables CS_SUMMARY, CS_CODE, and CS_ORIGIN.

Parameters (S-type):

Summary: Error/status summary.
See subroutine CSSET for details.

Code: This gives the command status in more detail.
See subroutine CSSET for deatils.

Origin: (optional) Error/status originator.
See subroutine CSSET for details.

Eventually there may be other parameters, so set the VL-bit.

Return codes:

0 – Status information retrieved successfully.
4 – Problems; bad parameter list; no VL bit, etc.

APPENDIX D
MACRO LIBRARY UTILITIES

There are two utility programs which may be of use in maintaining macro libraries. They are currently under the signon ID MAC.

LIBGEN: MACRO LIBRARY GENERATOR

The program MAC:LIBGEN is a simple program to generate or recreate a macro library. It is invoked with an MTS command of the form

$RUN MAC:LIBGEN [logical-units] [PAR=option]

The following logical units may be specified on the RUN command:

0	Specifies a file containing an existing macro library, or a sequence of macros.
1	Specifies a file in which a new macro library is to be built.
SERCOM	is used to display messages issued by the program.

The PAR field may specify either or both of the options

BUILDDIR	(abbreviation BUILD) indicates that the input from unit 0 has no directory, so one should be built from the information on the MACRO or FUNCTION lines in the input.
SORTDIR	(abbreviation SORT) means that the macro names in the output library should be sorted alphabetically, rather than preserving the order from the input directory.

The input on unit 0 is intended to be an existing macro library, possibly with extra macros that are not in the directory. The program finds all MACRO and FUNCTION lines in the input file, and uses the names from these to build the output library.

Line numbers in the input file are ignored completely. The order of all macros in the input directory, will be preserved in the output library (unless SORTDIR is specified).

LIBLIST: MACRO LIBRARY LISTER

The program MAC:LIBLIST is a simple program that can be used to produce a listing of a macro library. If a macro cannot fit in the space remaining on the current page, a new page will be started. It is invoked with an MTS command of the form

```
$RUN MAC:LIBLIST [logical-units] [PAR=option]
```

The following logical units may be specified on the RUN command:

SCARDS	specifies the macro library to be listed.
SPRINT	specifies the file or device on which the listing is to be produced.

The output is intended to be suitable for the Xerox 9700 in two-sided portrait mode. The page numbers and titles are alternated for front/back pages. The parameter ONESIDED may be specified in the PAR field to prevent alternation of page numbers.

APPENDIX E
EXAMPLES

Here are several examples of functions and macros, borrowed from actual user's macro libraries.

Two versions of functions to prompt the user:

```
>Function Confirmation Prompt_String
>*
>*  Description: Confirmation
>*  This function is used to confirm things.
>*  You say things like
>*  "if confirmation('Signoff now?'), emit '$Signoff'".
>*
define Attn_Flag
define Eof_Flag
define Response
write "&"||Prompt_String on *Msink*
read Response from *Msource*@UC attn->Attn_Flag eof->Eof_Flag prefix=" "

if not (Attn_Flag or Eof_Flag)
   if (Response = "Y") or (Response = "YES") or (Response = "OK")
   or (Response = "!") or (Response = "O.K."), exit True
endif
exit False
>Endfunction


>Function Prompt(where,what)
>*
>*  This macro writes out a prompt string on the unit designated
>*  by 'where' and reads a reply from it.  The prompt string is
>*  given by the 'what' parameter.
>*
>define response
>define Eof
write what on {where}
loop
  read response from {where} prefix='?' eof->Eof
  if Eof, exit FALSE
  set var response = uppercase(response)
  if response = 'OK' or response = 'YES' or response = 'O.K.'
      or  response = 'Y'
    exit TRUE
  elseif response = 'NO' or response = 'N'
    exit FALSE
  endif
  write 'Type OK or YES to confirm, NO to decline.' on {where}
endloop
exit False
>endfunction
```

A macro to create or empty a file:

```
>MACRO Crorem WhichFile Size="10p" "OK"
>*
>*  Crorem  --  Create or Empty a file.  Takes a file size and either
>*  creates it or empties it depending on its previous existence.
>*  If a confirmation is given, it passes it along to the empty
>*  command.
>*
DEFINE Confirmation=""

IF OK = "PRESENT", SET VAR Confirmation = "OK"
IF FILE(WhichFile) EXISTS
    $EMPTY {WhichFile} {Confirmation}
    $CONTROL {WhichFile} SIZE={Size}
ELSE
    $CREATE {WhichFile} SIZE={Size}
ENDIF
>ENDMACRO
```

A function to sort an array alphabetically:

```
Function Sort Arr
>*
>* Sorts the array "Arr" into ascending order.
>*
>* The algorithm is the original Shell sort from CACM 2,7 (1959)
>* as modified by Frank and Lazarus.
>*
Define N
Define M
Define I
Define J
Define K
Define Temp
Set var N = Max_Subscript(Arr)
Set var M = N
Loop while M>1
   Set var M = 2*Integer_Part(M/4)+1
   Set var K = N-M
   Set var J = 1
   Loop while J<=K
      Set var I = J
      Loop while I>0
         If Arr(I) <= Arr(I+M), Exitloop
         Set var Temp = Arr(I)
         Set var Arr(I) = Arr(I+M)
         Set var Arr(I+M) = Temp
         Set var I = I-M
      Endloop
      Set var J = J+1
   Endloop
Endloop
Exit Arr
>EndFunction
```

Extending commands to take lists:

The following two macros go together; the first calls the second.

```
>Macro cmd @etc
>*
>*  This macro scans its arguments for a parenthesised list and
>*  emits a series of commands--one for each of the items in
>*  parentheses.  The parenthesised item must be separated by at
>*  least one blank from the other text of the line.
>*
define parameters=' ' || parstring || ' '
define i
define j
define string
define text
>*
set var i=find_string(parameters(1...size(parameters)),' (')
set var j=find_string(parameters(i+1...size(parameters)),') ')
if i = 0 or j = 0
    Cmd_with_pat {"'"||parstring||"'"}
endif
>*  Have a parenthesised string--just one level deep--now break
>*  this up into component parts.
set var j = j + i
loop for string over parameters(i+1...j)
    if i = 1
        set var text=string
    else
        set var text=parameters(1...i) || string
    endif
    if j ¬= size(parameters)-1, set var text=
     text || parameters(j+1...size(parameters))
    Cmd_with_pat {"'"||text||"'"}
endloop
>endmacro


>Macro Cmd_with_pat Parameters
>*
>*  This macro checks its argument for containing a question mark.
>*  If it contains one, the question mark is expanded into a file
>*  name, and the string is emitted as a command.
>*
Define i
Define j
Define text
Define string
>*
>*  Check for question marks and assume that they are in a file name.
>*
   set var i=find_string(parameters(1...size(parameters)),'?')
   if i = 0
      emit parameters
      exit
   endif
>*
>*  Question mark found.  Search backwards from that point to
>*  locate first preceding blank
>*
```

```
   set var j=find_string(parameters(i...size(parameters)),' ')
   if j = 0
      set var i = size(parameters)
   else
      set var i = i + j - 2
   endif
   set var j=find_string(reverse(parameters(1...i)),' ')
   if j = 0
      set var j = 1
   else
      set var j = i - j + 2
   endif
   if j = 1
      define text_front = ''
   else
      define text_front = parameters(1...j-1)
   endif
   if i = size(parameters)
      define text_rear = ''
   else
      define text_rear = parameters(i+1...size(parameters))
   endif
   loop for string over files(parameters(j...i))
      set var text = text_front || ' ' || string || ' ' || text_rear
      emit text
   endloop
>Endmacro
```

Archiving by subject in the message-system:

The following three macros go together:

```
>Macro Ark "IN" Subject "NOTE" Note_Text@NOPROMPT
     Message="Active" Archive_File=CCID:Archive
>*  This macro is meant to be used in $MESSAGESYSTEM mode to archive
>*  messages according to a subject heading.  It uses the file
>*  "ARCHIVE" to maintain a directory of message subjects, which is
>*  read in the first time the macro is invoked.  Subsequently, the
>*  directory is added to as new subjects are incorporated.
>*
>*  Subject - A one word subject to classify the item under.  If it
>*     is not present in the directory, then the macro prompts for
>*     confirmation of the addition and allows a display of current
>*     subject categories.
>*  Note_Text - A header which is placed at the front of the message.
>*     It may be later scanned for using the editor.
>*  Message - An explicit message number which is to be archived.  If
>*     left off, this is the active message.
>*
If CLS_NAME ISNT "MESS"
    Write "Only in $MESSAGE..."
    Exit
Endif
>*
Define Subj_Pair
Define Subj_Item
Define Temp
```

```
Define Junk
Define String
>*
>*  See if the directory has been read yet during this invocation of
>*  the macro processor.  If not, read it in.
>*
Read_Subj_Index Archive_File={Archive_File}
>*
>*  Archive subjects are now read in.  See if the new subject is in
>*  the current directory of names.
>*
Define New_Index = Find_Subject(Subject)
If New_Index = 0, Exit
>*
>*  Now, the first line number of the subject category is in the
>*  variable New_Index.  Look in the file for a free message
>*  position for the message, making sure that the message does not
>*  exceed the range for it
>*
Define Flag
Loop for Temp from New_Index by 1 Until Temp > New_Index+999
    Read String FROM FDUB=Subj_Fdub EOF->Flag LNR=Temp MODS=INDEXED
    If Flag, Exitloop
Endloop
If NOT Flag
    Write " Sorry - No room in that subject category."
Else
    Define Msg_Index=Temp
    Emit "Display {Message} History=full
      Output={Archive_File}({Msg_Index},{Msg_Index}.999,.001)
      Header Subject Recipients Text"
    If Note_Text ISNT "" OR Note = "PRESENT" AND Note_Text IS ""
>*
>*  A note to append to the message.  Gyrations are undergone to
>*  put the note text following the header generated by the
>*  messagesystem.  This is done by scanning for the blank line
>*  which it follows the header with, and renumbering the file
>*  to make room for the note line.
>*
       Loop for Temp from Msg_Index+.001 by .001 to Msg_Index+.05
           Read String FROM FDUB=Subj_Fdub EOF->Flag LNR=Temp
                MODS=INDEXED
           If Flag OR String=" ", Exitloop
       Endloop
       If Flag OR String ISNT " "
           Write " Sorry - Couldn't append the note text"
           Write "   (Flag={Flag}, String='{String}', Lnr={Temp})"
           Exit
       Endif
       Define Note_Index=Temp
       Define Note_Line(10)
       Define Note_Lines
       If Note_Text IS ""
           Write " Note text:"
           Loop for Temp from 1 until Temp > 10
               Set var Note_Line(Temp)=""
               Read Note_Line(Temp) EOF->Flag
               If Flag or Note_Line(Temp)="", Exitloop
           Endloop
           Set var Note_Lines=Temp-1
       Else
```

```
            Set var Note_Line(1)=Note_Text
            Set var Note_Lines=1
        Endif
        Emit "$RENUMBER {Archive_File} {Note_Index} {Msg_Index+.999}
        {Note_Index+(Note_Lines/1000)} .001"
        If CS_Code ISNT 0
            Write " Sorry, couldn't put in the note text"
            Exit
        Endif
        Write " Note:  "||Note_Line(1) ON FDUB=Subj_Fdub
            LNR=Note_Index MODS=INDEXED
        Loop for Temp from 2 Until Temp > Note_Lines
            Write "   "||Note_Line(Temp) ON FDUB=Subj_Fdub
               LNR=Note_Index+(Temp-1)/1000 MODS=INDEXED
        Endloop
    Endif
Endif
>Endmacro


>Function Find_Subject(Subject)
If Variable("Subj_Index('" || Uppercase(Subject) || "')") doesnt exist
   Define Temp
   Define Junk
   Define String
   Define Junk2
   Loop
      Write "This subject category isn't defined. OK to define it?"
        ON *MSINK*
      Read String from *MSOURCE* EOF->Temp
      If Temp, Exit
      Set var String=UPPERCASE(TRIM(String))
      If String = "NO", Exit 0
      If String = "OK" or String = "YES", Exitloop
      If String = "LIST" or String = "?"
>*          LIST - Give a list of all the subject names
         Set var String = " Subjects:  "
         Loop for Temp over Subj_Names
            If Size(String) > 50
               Write String
               Set var String = "   "
            Endif
            Set var String = String || Temp || "  "
         Endloop
         Write String
      Endif
      If Variable("Subj_Index('" || Uppercase(String) || "')") exists
          Write "Already defined, try again"
          Nextloop
      Endif
   Endloop
   Define New_Index=-999
   Loop for Temp over Uppercase(Subj_Names)
      Set var Junk = Temp(1)
      If Subj_Index(Junk) > New_Index
         Set var New_Index={Subj_Index(Junk)}
      Endif
   Endloop
   Set var New_Index=Integer_Part((New_Index+1000)/1000)*1000+1
>*     Make up and write the new index out.
   Set var Temp="(" || Subject || "," || New_Index || ")"
```

```
   Set var Subj_Index(Uppercase(Subject))=New_Index
   If Subj_Names = "()"
      Set var Subj_Names="(" || Temp || ")"
   Else
      Set var Subj_Names=Subj_Names(1|Size(Subj_Names)-1) ||
          "," || Temp || ")"
   Endif
   Write Subj_Names ON FDUB=Subj_Fdub MODS=INDEXED LNR=0
   Set var Temp=" New subject '" || Subject || "' defined."
   Write Temp
Else
   Define New_Index=Subj_Index(Uppercase(Subject))
Endif
Exit New_Index
>Endfunction


>Macro Read_Subj_Index  Archive_File=
If VARIABLE("Subj_Index") DOESNT EXIST
   Define Subj_Index("Subject_Name") GLOBAL
   Define Subj_Names GLOBAL
   Define Subj_Fdub GLOBAL
   Define Temp
   Define Junk
   Define Subj_Pair
>*
>*  Read archive directory (at line 0) and build up a variable which
>*  contains the archive names.  The format of the directory is:
>*  "((subject,line#),(subject,line#),...)"
>*
   Read Subj_Names FROM {Archive_File} MODS=INDEXED LNR=0
     FDUB->Subj_Fdub EOF->Temp
   If Temp, Set var Subj_Names="()"
   Loop for Subj_Pair over Subj_Names
       Set var Junk = Uppercase(Subj_Pair(1))
       Set var Subj_Index(Junk) = Subj_Pair(2)+0
   Endloop
Endif
>Endmacro
```

APPENDIX F
MACRO DEFINITION AND EXPANSION

Here is a more detailed description of the macro definition and expansion process intended for someone interested in learning exactly how the macro processor has been hooked into MTS. The same "HASPLOG" example discussed earlier (in section O.1) is discussed in greater detail.

First a digression about how MTS does its I/O: Programs obtain data from *SOURCE* by calling the various I/O subroutines: READ, SCARDS, etc. Inside MTS all the I/O subroutines sit in the DSRI assembly (DSR-Interface). DSRI always obtains its data from DSRs: the file DSR, a terminal DSR, a tape DSR, etc. DSRI can nicely be divided into three parts:

(1) Prefix

check out the modifiers, open files if necessary, process line ranges, etc.

(2) DSR

Call the DSR to get the data (the FDUB says which DSR to call)

(3) Postfix

Clean up, do logging, return to the user.

It was mentioned earlier that when something is being read from *SOURCE*, MTS checks in with the macro processor before and after it gets a data line. Stated in these internal terms, MTS calls the macro processor from DSRI in the Prefix before just before calling the DSR, and it calls the macro processor in the Postfix just before returning to the user.

Let us look at the same example as before:

```
# set macros=on
# >macro hasplog
? $run unsp:hasplog
? 1 u w164
? stop
? >endmacro
#
# hasplog
# Execution begins
:  JOB: 674553  (4917317)  USERID:W164 S 07-08-82
:  JOB: 674559  (4917323)  USERID:W164 S 07-09-82
# Execution terminated
```

Here is what happens in all its gory detail:

(1) MTS (IN subroutine) calls SOURCIN to get a command.

(2) It gets “set macros=on” and invokes the $SET command which calls the macro processor’s “open” entry. Macro processing is now enabled.

(3) MTS (IN subroutine) calls SOURCIN to get another line.

(4) Macro processor call at DSRI-prefix: nothing to do.

(5) DSRI reads from the terminal DSR: “macro hasplog”.

(6) Macro processor call at DSRI-postfix: switch to macro definition mode and tell DSRI to go backto the prefix.

(7) Macro processor call at DSRI-prefix: nothing to do.

(8) DSRI reads from the terminal DSR: “$run unsp:hasplog”.

(9) Macro processor call at DSRI-postfix: insert “$run unsp:hasplog” as the 2nd line in macro HASPLOG. Tell DSRI to go back to prefix.

(10) Macro processor call at DSRI-prefix: nothing to do.

(11) DSRI reads from the terminal DSR: “l u w164”.

(12) Macro processor call at DSRI-postfix: insert “l u w164” as the 2nd line in macro HASPLOG. Tell DSRI to go back to prefix.

(13) Macro processor call at DSRI-prefix: nothing to do.

(14) DSRI reads from the terminal DSR: “stop”.

(15) Macro processor call at DSRI-postfix: insert “stop” as the 3rd line in macro HASPLOG. Tell DSRI to go back to prefix.

(16) Macro processor call at DSRI-prefix: nothing to do.

(17) DSRI reads from the terminal DSR: “endmacro”.

(18) Macro processor call at DSRI-postfix: recognize the ENDMACRO command and switch out of macro insertion mode. Tell DSRI to go back to the prefix.

(19) Macro processor call at DSRI-prefix: nothing to do.

(20) DSRI reads from the terminal DSR: "" (user typed a null line).

(21) Macro processor call at DSRI-postfix: nothing to do.

(22) Return from the SOURCIN call at (3) with a null line with the macro processor having defined macro HASPLOG in the interim.

(23) MTS ignores the null line and calls SOURCIN again.

(24) Macro processor call at DSRI-prefix: nothing to do.

(25) DSRI reads from the terminal DSR: “hasplog”.

(26) Macro processor call at DSRI-postfix: The macro invocation is recognized and the we switch to macro expansion mode. Tell DSRI to go back to the prefix.

(27) Macro processor call at DSRI-prefix: the first line of macro HASPLOG is produced: “$run unsp:hasplog”. Tell DSRI to skip the DSR-call.

(28) Macro processor call at DSRI-postfix: nothing to do (if there was another macro XXXX, and the line just produced was “XXXX” then then we’d push into expansion mode for XXXX and go back to DSRI-prefix; this is how recursion works.)

(29) Return from the SOURCIN call at (23) with “$run unsp:hasplog”.

(30) MTS starts up the $Run processor and loads up UNSP:HASPLOG. (Actually, first the macro processor checks to see if “run” is a macro, which it is not in this example.) This involves much traffic through DSRI, but since none of it involves the SOURCE-stream the macro processor is not invoked.

(31) UNSP:HASPLOG eventually calls SCARDS, which defaults to *SOURCE*, to get a command.

(32) Macro processor call at DSRI-prefix: it is still in macro expansion mode and retrieves line two of the HASPLOG macro: “l u w164”. Tell DSRI to skip the DSR call.

(33) Macro processor call at DSRI-postfix: nothing to do.

(34) Return from SCARDS call at (31) with: “l u w164”.

(35) UNSP:HASPLOG processes the LOCATE command printing out two lines on SPRINT.

(36) UNSP:HASPLOG calls SCARDS for another command line.

(37) Macro processor call at DSRI-prefix: return next line of the macro expansion: “stop”. Tell DSRI to skip the DSR call.

(38) Macro processor call at DSRI-postfix: nothing to do.

(39) Return from SCARDS call at (36) with: “stop”. Since the macro processor has just given out the last line in macro HASPLOG, it switches out of macro expansion mode.

(40) UNSP:HASPLOG processes the STOP command and unloads.

(41) The $Run command unloads the EXEC CLS, prints the “Execution Terminated” message, and returns to IN, then IN calls SOURCIN to get another command; ...and off we go again...

This level of detail is probably more than most users are interested in, but for those who have been wondering how macro expansion works, it is hoped this trace has clarified things a bit. It looks like an

awful lot of work just to process a simple macro, but if you go to a low enough level with anything, it will look complicated.

APPENDIX G
INPUT TIME CONVERSION

The ABSTIME and RELTIME built-in functions, if given a string as an argument, interpret that string as specifying a time in terms that people normally use. Following are the rules for that specification, extracted from the document UBC PLUS TIME. Any of the forms below may be used as input to ABSTIME; only the forms under "Time Intervals" may be used as input to RELTIME.

Input Time Components

Time/date input conversion is driven by a grammar that attempts to accept most styles of time/date presentation that are commonly used. It generally requires that the components specified be meaningful and unambiguous.

An input time may be composed of the following pieces. They may be put together in most meaningful combinations, using commas and blanks as conventional separators.

Year — A year is a two-digit number in the range 32 to 99, inclusive, or a four-digit number in the range 1900 to 1999, inclusive. In some unambiguous contexts, a two-digit number in the range 0 to 31 will also be recognized as a year.

Month — A month is one of "January", "February", ..., "December". Month names may be abbreviated to their first three characters.

Day — The day of the month is specified by a number in the range 1 to 31, inclusive.

Time — Times are integer strings containing ":" and/or "." and/or followed by "AM" or "PM" (or "A.M." or "P.M.") A fully specified time appears as "23:22:05.67835".

The time must specify the hours and minutes, with the seconds and microseconds components optional. The minutes component may also be omitted if "AM" or "PM" appears.

Thus, the following are all allowed:

12 AM
23:59
00:01:35.5

Day of Week — Days of the week are specified as "Monday", "Tuesday", ..., "Sunday". They may be abbreviated to the first three characters of the day name ("Mon", "Tue", etc.).

The day-name is just checked for correctness with respect to the other date

components. Currently, a day-name cannot be used by itself.

MM–DD–YY

For compatibility with older date input and output routines the MM–DD–YY form of month, day and year is supported (it may also be specified as MM/DD/YY). For example, "11-06-79" is November 6, 1979.

This form is not recommended because of its ambiguity. Depending on where you live, the convention may be MM–DD–YY, DD–MM–YY, or YY–MM–DD. This package supports the MM–DD–YY form because that form has been the MTS standard.

IBM

For compatibility with certain IBM software, the form YY.DDD is supported, where YY is a two- or four-digit year number and DDD is the day number in the year. For example, "79.103" is April 13, 1979.

Miscellaneous Forms

The following strings are recognized, with the obvious meanings.

TODAY
YESTERDAY
TOMORROW
NOW

The first three forms represent a date only, and may be combined with a time specification. The form “NOW” represents the current time (to the microsecond level).

Time Intervals

Times may alternatively be specified in terms of a number of time units in the future from the present time (for ABSTIME) or in terms of a number of time units without a base (for RELTIME). This specification is of the form

<integer> <unit>

where “unit” may be one of

MICROSECONDS
MILLISECONDS
SECONDS
MINUTES
HOURS
DAYS
WEEKS
MONTHS
YEARS

and some appropriate abbreviations.

For arguments to RELTIME, since the base is not known, a month is assumed to be 30 days and a year 365 days.

For example, if the specification "2 weeks" is given when the time is 15:40:06.23572 January 4, 1984, the result will be "15:40:06.23572 January

18, 1984".

Time-zone	A time-zone specification may be any of the standard three-letter time-zone codes defined in the MTS system table of time-zone names. “GMT”, which is not in that table, is also accepted.

Defaults

It is not necessary to specify all the components of the time (year through microsecond). The input conversion routines will default unspecified components in many cases.

When converting, the defaulting is performed with respect to the current time. Any unspecified low-order components are always set to the beginning of the interval. Unspecified high-order components are copied from the current time. For example, if it is currently any time in 1984, then "Jan 25" is equivalent to "00:00:00.0 Jan 25/84". However, low-significance *date* components will not be defaulted. That is, "Oct 79" is not valid as a single time; the user must specify the day.

APPENDIX H
ADDITIONAL BUILT-IN FUNCTIONS

The following built-in functions were added in June 1990.

BITAND, BITNOT, BITOR

Syntax
```
integer = BITAND(integer,integer)
integer = BITOR(integer,integer)
integer = BITNOT(integer)
```

The BITAND, BITOR, and BITNOT functions provide bitwise-logical functions. The arguments must be only integer (or long integer); to process operands of other types use the USE_AS function.

EXTERNAL

Syntax
```
>SET VAR x = EXTERNAL("EDIT","y")
>SET VAR EXTERNAL("EDIT","y") = value
```

The first form is used to get the value of the MTS File Editor's variable "y" and put it into "x". The second form is used to set the value of the File Editor's variable "y" to "value". The first argument must currently be "EDIT" and it specifies the editor CLS invocation. Any editor variables to be set must already exist (they will not be created automatically). The only two predefined editor variables that can be set are ED_VRULER and ED_WORK. When getting an editor variable, the variable must be of integer, line-number, or string types. When setting an editor variable, the value given must be coercible into an integer (not a long integer), a line number, or a string.

HEX

Syntax
```
string = HEX(x)
```

The HEX function provides a means of displaying the actual bit patterns (in hex) of the primitive

data types used by the macro processor.

The value specified as argument must be of type integer (or long integer), line-number, Boolean, abstime, reltime, or string (or the internal type called hex).

For example,

```
HEX("ABC")
```

has value “C1C2C3”.

INTEGER_FORMAT

Syntax

```
string = INTEGER_FORMAT(integer,string)
```

The INTEGER_FORMAT function converts the integer first argument to a string, formatted according to the characters in the second string. See Appendix I for the definition of the formatting string.

For example,

```
INTEGER_FORMAT(42,"999")
```

has value “042”.

SHIFT_LEFT, SHIFT_RIGHT

Syntax

```
y = SHIFT_LEFT(x,amount)
y = SHIFT_RIGHT(x,amount)
```

where “x” must be an integer (or long integer) or a line number and is the quantity to be shifted. “y” will be of the same type as “x”. “amount” must be an integer and is the number of bits to shift. This is a logical shift, not an arithmetic shift. If the shift amount is negative, the shift goes in the reverse direction.

TIME_FORMAT

Syntax

```
string = TIME_FORMAT(absolute time)
string = TIME_FORMAT(absolute time,string)
string = TIME_FORMAT(absolute time,integer)
string = TIME_FORMAT(absolute time,string,string)
string = TIME_FORMAT(absolute time,integer,string)
```

The TIME_FORMAT function converts the time first argument to a string, formatted according to characters in the second (string) argument. If the second argument is missing, an integer 0 is assumed.

If the second argument is an integer in the range 0 to 7, then those represent eight predefined time patterns. The optional third argument is a time zone, if the time is to be converted according to a time zone different from the one currently in effect.

When the second argument is a string, it is copied to the output string with various substitutions being performed. Substitutions are indicated in the string by codes surrounded by "<" and ">". See Appendix J for a description of the codes.

Examples:

```
# >dis value(time_format(abstime("now"),0))
#  time_format(abstime("now"),0): 23:00:40.989436 Sun Apr 1/90 EDT
# >dis value(time_format(abstime("now"),0,"est"))
#  time_format(abstime("now"),0,"est"): 22:01:09.497042 Sun Apr 1/90 EST
```

TIME_WAIT

Syntax

```
integer = TIME_WAIT(reltime)
integer = TIME_WAIT(abstime)
```

The TIME_WAIT function provides a means of doing a real-time wait. The "value" of this function is currently the return code from the system routine TWAIT, which should be zero.

Examples:

```
SET VAR X=TIME_WAIT(RELTIME("10 SECONDS"))
SET VAR X=TIME_WAIT(ABSTIME("11:30pm Jul 13"))
```

Warning: There is currently no way to interrupt out of a real-time wait; an attention interrupt will not take effect until after the time wait is up.

TRANSLATE

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = TRANSLATE(string,"ASCEBC")                                      │
│  string = TRANSLATE(string,"EBCASC")                                      │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

The string argument is translated according to the system translate tables, from ASCII to EBCDIC or from EBCDIC to ASCII, respectively.

USE_AS

```
Syntax──────────────────────────────────────────────────────────────────────┐
│                                                                           │
│  string = USE_AS(x,"STRING")                                              │
│  integer = USE_AS(x,"INTEGER")                                            │
│  linenumber = USE_AS(x,"LINENUMBER")                                      │
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
```

The USE_AS function provides a means of type-cheating. It does no conversion of its arguments, but just tells the macro processor to treat them as if they had a different type. This allows you, for example, to give a 4-character string as a "line number" to a READ or WRITE, or to take apart pieces of sense data (the SENSE function returns its result as a string, although parts of that "string" are usually integers, etc.).

The original type of "x" must be either integer (or long integer), line number, or string. If a string "x" is to be treated as a line number, the string must be 4 characters long. If a string "x" is to be treated as an integer, the string must be from 0 to 8 characters long: a zero-length string has value 0; strings of lengths 2, 4, or 8 are treated as signed integers (or long integers); strings of lengths 1, 3, 5, 6, or 7 characters are unsigned integers, right-justified with leading zeros.

APPENDIX I
INTEGER PICTURE FORMATS

For conversion of integers from internal form to printable form using the INTEGER_FORMAT built-in function, the form of the output string is described by a "picture" specification similar to those of COBOL or PL/1.

The number is first converted to a sequence of digits, which is then edited under control of the picture. Pictures can be used to effect scaling, left-or-right zero suppression, comma and decimal point insertion, and fixed or varying field width of the converted item.

Pictures

A picture is a sequence of characters describing the format desired for the converted string. The characters forming the picture may be any of the following:

9	specifies the position is to be occupied by a digit.
blank	used in place of "9" to indicate replacement of leading or trailing zeros with blanks.
Z, z	used in place of "9" to indicate suppression of leading or trailing zeros.
.	specifies literal insertion of a ".", if the position is followed by a digit. That is, decimal does not appear if it is passed over by right zero suppression. If it is preceded or followed by a blank, it will be replaced by a blank.
D, d	specifies literal insertion of a ".", even if there is no following character.
,	specifies literal insertion of a comma into a sequence of digits; no comma is inserted if the position in the output string is not both preceded and followed by a digit. If this position is preceded or followed by a blank, it will be replaced by a blank.
V, v	indicates the position at which to align the decimal point of the number being converted (i.e., the right-hand end of the number). If this is omitted, it is assumed to be at the right-hand end of the picture. The "V" has the effect of scaling the value.
P, p	is used to allow a "V" to appear past the last digit character of the picture. "P"s may appear only at the right of the picture. They have the effect of discarding the rightmost digits.

A valid picture may have a format like

(Z)(blank)(9)[.(9)(blank)(Z)(P)]

where (?) indicates 0 or more occurrences of the "?", and everything in [ ] is optional. Commas may

appear anywhere in the picture, and a "D" may appear instead of ".". One "V" may appear, with the restriction that "Z", blank, and "P" are not allowed to the right of the "V".

If the number is negative, a sign will be placed in the rightmost unused "Z" or " " position left of the decimal. The pattern must provide at least one "Z" or blank if the number might be negative.

Examples

The following table indicates the results returned for some possible combinations of value and picture.

Value	Picture	Result
123456	"999999"	"123456"
123456	"ZZZZZ9"	"123456"
123456	"ZZ,ZZ9.99"	"1,234.56"
123456	" 9.ZZZ"	" 123.456"
123456	"ZZZ .ZZZ"	"123.456"
123456	""	See note 1
123456	"ZZZZ"	See note 1
123456	" .PPPV"	" 123 "
0	"999999"	"000000"
0	"ZZZZZ9"	"0"
0	"Z,ZZZ,9.99"	"0.00"
0	" 9.ZZZ"	" 0"
0	"Z .ZZZ"	" "
0	""	""
0	"ZZZZ"	""
0	" .PPPV"	" "
-1	"999999"	See note 2
-1	"ZZZZZ9"	"-1"
-1	"Z,ZZZ,9.99"	"-0.01"
-1	" 9.ZZZ"	" -0.001"
-1	"Z .ZZZ"	"-.001"
-1	""	See note 1
-1	"ZZZZ"	"-1"
-1	" .PPPV"	" - "
3000	" 9.999"	" 3.000"
3000	" 9. "	" 3 "
3000	"ZZZZZZZ9DZZZ"	"3."
3000	"ZZZZZZZ9.ZZZ"	"3"
-123	"999V"	See note 2
-123	"999"	See note 2
-123	" 999"	"-123"
-123	"Z999"	"-123"

Note 1: Produces the error message "Number too big for picture".

Note 2: Produces the error message "No room for sign".

APPENDIX J
TIME PICTURE FORMATS

For the TIME_FORMAT built-in function, the second argument is either a string giving a format or else an integer specifying a predefined format.

STRING second argument

When the format is directly given, it is a string (usually a constant), which is copied to the output string with various substitutions being performed. Substitutions are indicated in the string by codes surrounded by "<" and ">". If a "<" is to be copied to the output string, it appears in the format as "<<". The codes used to indicate substitutions are defined below.

Conversion Codes

Each substitution in the format string is indicated by an element of the form

<item,width,type>

where all but the first component are optional.

"item" is a code for the time/date element to be substituted. It may be one of:

DAY	day number.
MONTH	month.
YEAR	year.
DAYNAME	day name (i.e., Monday, Tuesday, etc.).
DAYOFYEAR	day number relative to the beginning of the year.
HOUR24	hour in 24-hour (number from 0 to 23) form.
HOUR12	hour in 12-hour format (number from 1 to 12).
AMPM	"am" or "pm" as appropriate.
MINUTES	minutes.
SECONDS	seconds.
MICROSECONDS	number of microseconds.

TIMEZONE time-zone code.

FTZ time-zone spelled out.

"width" indicates an output length for the converted value. It will be padded to at least this length. Padding may be on left or right and may be with blanks or zeros, depending on the item being converted and the conversion type. The default is generally the minimum number of positions in which the value will fit. Some items will be truncated if the converted value is longer than "width". Each item has a minimum below which it will not be truncated. Month and day names may be truncated to three or more characters; years may be truncated to two characters. Time zones may be truncated to two characters.

"type" indicates the type of output desired; it may be one of

NUMERICZERO The item is to be output as an integer with zero-padding to the field width (e.g., January as "1" or "01").

NUMERICBLANK The item is to be output as an integer with blank-padding to the field width (e.g., January as "1" or " 1").

ALPHABETIC The item is to be output as mixed-case alphabetic (e.g., January as "January").

UCALPHABETIC The item is to be output as uppercase only alphabetic (e.g., January as "JANUARY").

The default depends on the item being converted. It is NUMERICZERO for all time components, NUMERICBLANK for day and year, and ALPHABETIC for AMPM, DAYNAME, MONTH and TIMEZONE.

Example:

The standard MTS format could be represented by the format string

```
"<H>:<M>:<S>.<MIC> <DN> <MO> <D>, <Y>"
```

which gives

```
15:41:32.123456 Monday December 2, 1982
```

INTEGER second argument

When the second argument is an integer in the range 0 to 7, it specifies use of one of the following eight predefined formats:

value 0:

The format string is

```
"<H>:<M>:<S>.<Mic> <Dn,3> <Mo,3> <D>/<Y,2> <Tz>"
```

which produces output in the form

```
23:00:40.989436 Sun Apr 1/90 EDT
```

value 1:

The format string is

```
"<Y,2> <Mo,3> <D> <H>:<M>:<S> <Tz>"
```

which produces output in the form

```
90 Apr 1 23:01:25 EDT
```

value 2:

The format string is

```
"<Y,2>.<Dy,3> <H>:<M>:<S> <Tz>"
```

which produces output in the form

```
90.091 23:01:41 EDT
```

value 3:

The format string is

```
"<H>:<M>:<S> <Dn,3> <Mo,3> <D>/<Y,2> <Tz>"
```

which produces output in the form

```
23:01:53 Sun Apr 1/90 EDT
```

value 4:

The format string is

```
"<H>:<M>:<S> <TZ>"
```

which produces output in the form

```
23:02:05 EDT
```

value 5:

The format string is

```
"<DN,3> <MO,3> <D,2>/<Y,2>"
```

which produces output in the form

```
Sun Apr  1/90
```

value 6:

The format string is

```
"<H>:<M>:<S> <TZ>, <DN,3> <MO,3> <D,2>/<Y,2>"
```

which produces output in the form

```
23:02:24 EDT, Sun Apr  1/90
```

value 7:

The format string is

```
"<H>:<M>:<S> <FTZ>"
```

which produces output in the form

```
23:02:44 Eastern Daylight Time
```

Note: For values 0, 1, and 3, “<y,2>” will be “<y,4>” if the year is >2000.

APPENDIX K
SUBROUTINES TO ACCESS MACRO VARIABLES

There are two subroutines available to get and set the value of macro variables. The subroutines are actually designed for any variables, not just macro variables, but the current implementation only handles macro variables. The names are GETEXVAR (alternate name is GTXVAR) and SETEXVAR (alternate name is STXVAR), for "Get/Set External Variable". These are gated (user-callable) entries. Note that if the set entry is called for a variable that does not exist, it will create one and then set its value as requested.

The calling sequences are:

GETEXVAR(who,varname,varnamel,vartype,varvalue,varvaluel)

SETEXVAR(who,varname,varnamel,vartype,varvalue,varvaluel)

where

who — Must be either the 4-character string "MTS " or else a fullword binary zero, both of which specify the macro processor variables. (In the future, this may be allowed to be something else, e.g., "EDIT", to specify a different set of variables.)

varname — Location of first character of variable name.

varnamel — Location of fullword length of variable name.

vartype — Location of fullword integer specifying the type of the value:

1 means integer
2 means linenumber
3 means string
4 means Boolean

GETEXVAR: This parameter will be set by the subroutine to indicate what it put into the parameter "varvalue".

SETEXVAR: User must set this parameter before the call to indicate what "varvalue" contains.

varvalue — GETEXVAR: Location of the first byte of where the value is to be placed.

SETEXVAR: User must set the new value starting at this location.

varvaluel — Location of fullword length of variable value:

GETEXVAR: User must set this to the length of the area provided in "varvalue". For integers, it must be 1, 2, 4, or 8; for line numbers, it must be 4; for Boolean, 1 or 4. For strings, upon return from the subroutine it will have been set to the actual length returned. If the area is not big enough, it will be set to the length required and a return code of 16 given.

SETEXVAR: User must set this to the length of the value provided in "varvalue". For integers, it must be 1, 2, 4, or 8; for line numbers, it must be 4; for Boolean, 1 or 4; for strings, the length.

The return codes are:

0	Successful return.	
4	GETEXVAR:	Variable not found.
	SETEXVAR:	Variable not found and it was unable to create a global variable of that name (i.e., name syntactically bad).
8	GETEXVAR:	Value not exportable (i.e., it cannot be converted to integer, linenumber, Boolean, or string).
	SETEXVAR:	Type code is less than 1 or greater than 4.
12	GETEXVAR:	Not allowed to get value of that variable.
	SETEXVAR:	Not allowed to set value of that variable.
16	GETEXVAR:	Length in "varvaluel" not big enough. "varvaluel" has been set to length required.
	SETEXVAR:	Cannot get enough space to set variable to value that large.
20	Entity named by "who" parameter (i.e., the macro processor) is not active.	
24	Value of "varvaluel" parameter is unreasonable (i.e., negative or greater than 32767).	
28	At least one of the parameters is unaddressable or (for parameters the subroutine has to change) is in a protected (unchangeable) location.	

APPENDIX L
SYMBOL INDEX

– operator....51

¬= operator....52, 53

... operator....51

.AND. operator....53
.EQ. operator....52
.GE. operator....52
.GT. operator....52
.IS. operator....52
.ISNT. operator....52
.LE. operator....52
.LT. operator....52
.NE. operator....52
.NOT. operator....53
.OR. operator....53

< operator....52
<= operator....52

+ operator....51
+= operator....53

| | operator....51
| |= operator....53

* operator....51

/ operator....51

> operator....52
>= operator....52

: operator....55

@CHECK....89
@CLS....80
@ETC....82, 84
@NOPROMPT....82
@PKEY....80
@PROMPT....82
@RAW....85

@RECURSIVE....80
@SYSTEM....80

= operator....52, 53

ABS....57
ABSENT....83
ABSTIME....57
AND operator....53
ARRAY....50, 58
ATTN....98

BATCH....69
BITAND....121
BITNOT....121
BITOR....121
BY....76

CAN_CONVERT....58
CLS_NAME....70
CODE....88
CONTROL....58
COST....59
CPU_TIME....70
CS_CODE....70
CS_ORIGIN....70
CS_SUMMARY....70
CUINFO....59

DATE....70
DEFINE....45, 46, 47
DISPLAY....50, 51
DOES operator....53
DOESNT operator....53
DOUBLE....60
DUPL....60

ELSE....74
ELSEIF....74
EMIT....97
ENDFILE....98
ENDFUNCTION....93
ENDIF....74
ENDLOOP....75
ENDMACRO....81
ETC....82, 84
EVAL....60
EXIST operator....53
EXISTS operator....53
EXIT....88
EXITLOOP....77
EXTERNAL....121

FILE....61
FILE_INFO....61
FILES....62
FIND_CHAR....62
FIND_STRING....63
FOR....76
FORGET....48
FROM....76
FULL_SCREEN_POSSIBLE....70
FUNCTION....93
FUNCTION_NAME....85
FUNCTIONS....50

GENERATE....98
GETEXVAR....131
GOTO....74
GUINFO....63

HEX....121
HOST_NAME....70

IF....74
IGNORE_CHAR....63
INSTALLATION....70
INSTALLATION_CODE....71
INSTALLATION_NAME....71
INTEGER_FORMAT....122, 125
INTEGER_PART....64
IS operator....52
ISNT operator....52

LABEL....73
LAST_SIGNON....71
LOCK....64
LOOP....75
LOWERCASE....64
LPAD....64
LTRIM....65

MACLIB....50, 95
MACRO....50, 81
MACRO_NAME....85
MACROECHO....89
MACROPROMPT....82
MACROS....50
MACROTRACE....89
MAX....65
MAX_SUBSCRIPT....65
MICROSECONDS....65
MIN....66
MIN_SUBSCRIPT....66

NBR_POSITIONAL_PAR 82, 85
NEGATED 83
NEXTLOOP 77
NONNULL 53, 71
NOT operator 53
NULL 53, 71, 98

OR operator 53
OVER 76

PARSTRING 82, 85
POSITIONAL_PAR 82, 84
PRESENT 83
PROJECT 71

QUOTE 66

RATE 71
RAW 85
RC 79
READ 90
RELTIME 66
REPLACE 67
REVERSE 67
RPAD 67
RUNRC 71

SENSE 68
SET 49, 89, 95
SETEXVAR 131
SHIFT_LEFT 122
SHIFT_RIGHT 122
SIGNONID 71
SIZE 68
STRUCTURE 50
SUBSTITUTE 68
SYMBOLS 51
SYSTEM_VARIABLES 50

TASKNBR 71
TERMTYPE 71
TIME 72
TIME_FORMAT 122, 127
TIME_WAIT 123
TO 76
TRANSLATE 124
TRIM 68
TRUE 56

UNTIL 76
UPPERCASE 69
USE_AS 124

USER_VARIABLES....50

VALUE....50
VAR....49
VARIABLE....69
VARIABLES....50

WHILE....76
WRITE....92

INDEX

–, 72
– operator, 51

¬=, 72, 73
¬= operator, 52, 53

flag character, 43
field selection, 55
... operator, 51

.EQ. operator, 52
.GE. operator, 52
.GT. operator, 52
.IS., 52
.IS. operator, 52
.ISNT., 52
.ISNT. operator, 52
.LE. operator, 52
.LT. operator, 52
.NE. operator, 52

<, 72
< operator, 52
<=, 72
<= operator, 52

+, 72
+ operator, 51
+=, 73
+= operator, 53

| |, 72
| | operator, 51
| |=, 73
| |= operator, 53

&(, 48
&), 48

*, 72
* operator, 51
>, 43
*C_, 89

*CMDMACLIB, 96

/, 72
/ operator, 51
/END, 96

>, 72
> operator, 52
>*, 77
>=, 72
>= operator, 52

:, 47, 72
: operator, 55

@CHECK, 89
@CLS, 80
@ETC, 82, 84
@MACRO, 100
@MACRO_FLAG_REQUIRED, 100
@MFR, 100
@NOMACRO, 79, 100
@NOMACRO_FLAG_REQUIRED, 100
@NOMFR, 100
@NOPROMPT, 82
@PKEY, 80
@PROMPT, 82
@RAW, 85
@RECURSIVE, 80
@SYSTEM, 48, 80

=, 72, 73
= operator, 52, 53

{, 48

ABS, 57
ABSENT, 83, 84
ABSTIME, 57
addition, 51
aggregates, 46
AND, 72
AND operator, 53
array, 46, 50, 58
array denotations, 57
array subscription, 54
attention interrupt, 91, 93, 98
ATTN, 91, 93, 98

BATCH, 69
BITAND, 121
BITNOT, 121

BITOR, 121
Boolean, 46
Boolean constants, 56
BY, 76

CAN_CONVERT, 58
character constants, 56
character string, 46
CLS, 48
CLS_NAME, 70
CODE, 88
comments, 77
comparatives, 52
concatenation, 51, 72
CONSTANT, 45, 47
constants, 56
continuation lines, 78
CONTROL, 58
COST, 59
CPU_TIME, 70
CS_CODE, 70, 88
CS_ORIGIN, 70
CS_SUMMARY, 70, 88
CUINFO, 59

DATE, 70
definitional modifier, 82
DEFINE, 45, 46, 47
DISPLAY ARRAY, 50
DISPLAY FUNCTIONS, 50
DISPLAY MACLIB, 50
DISPLAY MACRO, 50
DISPLAY MACROS, 50, 96
DISPLAY STRUCTURE, 50
DISPLAY SYMBOLS, 51
DISPLAY SYSTEM_VARIABLES, 50, 69
DISPLAY USER_VARIABLES, 50
DISPLAY VALUE, 50
DISPLAY VARIABLES, 50
division, 51
DOES operator, 53
DOESNT operator, 53
DOUBLE, 60
DUPL, 60

EDIT, 97
ELSE, 74
ELSEIF, 74
EMIT, 97
end-of-file, 91, 96, 98
ENDFILE, 98
ENDFUNCTION, 93

ENDIF, 74
ENDLOOP, 75
ENDMACRO, 81
EOF, 91
equal to, 52
ERR, 91, 93
ERRMSG, 91, 93
error messages, 43
error return, 91, 93
ERRRC, 91, 93
ETC, 82, 84
EVAL, 60
EXIST operator, 53
EXISTS operator, 53
EXIT, 88
EXITLOOP, 77
EXPLAIN, 42
expression, 55
EXTERNAL, 121

FALSE, 56
FDUB, 90, 91, 92, 93
FILE, 61
FILE_INFO, 61
FILES, 62
FIND_CHAR, 62
FIND_STRING, 63
FOR, 73, 76
FORGET, 48
FROM, 76, 90
FULL_SCREEN_POSSIBLE, 70
FUNCTION, 93
function call, 54, 72
FUNCTION_NAME, 85
FUNCTIONS, 50

GENERATE, 98
GETEXVAR, 131
GLOBAL, 45
GOTO, 74
greater than, 52
greater than or equal to, 52
GUINFO, 63

HEX, 121
hexadecimal input, 56
HOST_NAME, 70

IF, 74
IGNORE_CHAR, 63
in-line macros, 96
input, 90

INSTALLATION, 70
INSTALLATION_CODE, 71
INSTALLATION_NAME, 71
integer, 46
integer constants, 56
INTEGER_FORMAT, 122, 125
INTEGER_PART, 64
IS, 72
IS operator, 52
ISNT, 72
ISNT operator, 52
iteration, 75

}, 48

LABEL, 73
LAST_SIGNON, 71
less than, 52
less than or equal to, 52
LIBGEN, 105
LIBLIST, 105
line number, 46
line-number constants, 56
LINENUMBER, 91, 93
LNR, 91, 93
LOCK, 64
logical connectives, 53
LOOP, 75
LOOP FOR, 73
LOWERCASE, 64
LPAD, 64
LTRIM, 65

MAC:LIBGEN, 105
MAC:LIBLIST, 105
MACLIB, 50, 80, 95
MACRO, 50, 81
macro flag character, 43
macro library, 95
MACRO_NAME, 85
MACROECHO, 89
MACROPROMPT, 82
MACROS, 50
MACROTRACE, 89
MAX, 65
MAX_SUBSCRIPT, 65, 86
MICROSECONDS, 65
MIN, 66
MIN_SUBSCRIPT, 66
MODIFIERS, 90, 92
MODS, 90, 92
multiplication, 51

NBR_POSITIONAL_PAR, 82, 85
NEGATED, 83
negation, 51
NEXTLOOP, 77
NONNULL, 53, 71
NOT, 72
not equal to, 52
NOT operator, 53
NULL, 53, 71, 98
numeric operators, 51

ON, 92
OR, 72
OR operator, 53
output, 92
OVER, 76

PARSTRING, 82, 85
POSITIONAL_PAR, 82, 84
PREFIX, 91, 93
PRESENT, 83
PROJECT, 71

QUOTE, 66

RATE, 71
RAW, 85
RC, 57, 59, 63, 79
READ, 90
RELEASE, 91, 93
RELTIME, 66
REPLACE, 67
return code, 57, 59, 63, 79
REVERSE, 67
RPAD, 67
RUNRC, 71

scope, 47
SENSE, 68
SET, 89
SET VAR, 49, 95
SETEXVAR, 131
SHIFT_LEFT, 122
SHIFT_RIGHT, 122
SIGNONID, 71
SIZE, 68
SQD, 48, 49
string operators, 51
string substitution, 48, 55
structure, 46, 50
subscription, 72
SUBSTITUTE, 68

substitution, 53, 73
substring, 51, 72
subtraction, 51
SYMBOLS, 51
SYSTEM_VARIABLES, 50, 69

table, 46
TASKNBR, 71
TERMTYPE, 71
TIME, 72
TIME_FORMAT, 122, 127
TIME_WAIT, 123
TO, 73, 76
TRANSLATE , 124
TRIM, 68
TRUE, 56
types, 46

UNTIL, 76
UPPERCASE, 69
USE_AS, 124
USER_VARIABLES, 50

VALUE, 50
VAR, 49
VARIABLE, 69
variables, 45, 50, 55

WHILE, 76
WRITE, 92

00000000, 96

Reader's Comment Form

MTS Command Extensions and Macros

Revised February 1991

Errors noted in publication:

Suggestions for improvement:

Date _____

Name _____

Address _____

Your comments will be greatly appreciated. Please fold the completed form as shown on the reverse side, seal or staple, and drop in Campus Mail or in the Suggestion Box at any Campus Computing Site.

fold here

Documentation Group
ITD Consulting and Support Services
The University of Michigan
611 Church, 2nd Floor
Ann Arbor, Michigan 48104-3056
USA

---

fold here